Implementing an event camera region of interest filter for real-time object tracking

A thesis submitted in part fulfilment of the degree of
Bachelor of Engineering (Honours)

by

Lachlan Spencer

U6672529

Supervisor: Prof. Jochen Trumpf

Examiner: Prof. Robert Mahony

Australian National University

College of Engineering and Computer Science
The Australian National University

June 2024

This thesis contains no material which has been accepted for the award of any other degree or diploma in any university. To the best of the author's knowledge, it contains no material previously published or written by another person, except where due reference is made in the text.

Lachlan Spencer
6 June 2024

©Lachlan Spencer

Acknowledgements

Thank you to my family for their constant support throughout my studies. Thank you to my partner Maggie for all her hard work keeping me fed and motivated.

Abstract

Reducing the amount of data processed in event camera systems lowers power consumption and increases the speed of computation of tracking algorithms. A filter that reduces the Region of Interest of the camera reduces the amount of data that is output and consequently processed, at the cost of performance. This performance decrease comes from the FPGA not transmitting events individually over USB, but in batches called buffers. By reducing the buffer size, the decreased performance can be negated. This leads to a decrease in data output with maintained system performance.

Contents

List of Figures

List of Tables

Nomenclature

RDK	Reference Design Kit
EVK	Evaluation Kit
FPGA	Field Programmable Gate Array
μs	microsecond
ROI	Region of Interest
ΔT_{event}	Time between events
ΔT_{buffer}	Time between buffers of events
RMSE	Root Mean Squared Error
RTC	Real Time Clock
WSL	Windows Subsystem for Linux
csv	Comma-Separated Values
GUI	Graphical User Interface
IC	Integrated Circuit
HDL	Hardware Description Language
PLM	Platform Loader and Manager
u-boot	Universal Bootloader
FSBL	First Stage Bootloader
OS	Operating System
MPSoc	MultiProcessor System On Chip
GPIO	General-Purpose Input/Output
MIPI	Mobile Industry Processor Interface

Chapter 1

Introduction

Technology evolves over time as its limitations can no longer keep pace with innovation. Camera technology is at this stage currently, as conventional frame-based sensor technology fails to keep up with new system requirements. As systems become faster and more efficient, technology needs to keep pace. Industries such as mining, Defence, and manufacturing, and fields such as robotics and autonomous vehicles use systems and devices that operate as close to real-time as possible. In most robotics applications, real-time is a term used when the system has a time performance requirement in the order of microseconds. Every process that the system performs takes a finite amount of time, meaning that for a system to perform in real-time, every process must be as efficient as possible.

Frame-based cameras have been the standard technology for a long time, however they come with one major drawback: the amount of data they produce. This problem worsens as the resolution of the camera increases. A standard 12 mega-pixel (MP) camera with a frame rate of 30 frames per second (fps) over one second produces 360 million pixels worth of data. This is an enormous amount of data that requires processing, and as resolutions improve, this only worsens. This is where newer technologies, such as event cameras, come in. Unlike frame-based cameras which operate by using a shutter that exposes all the pixels in the camera to a scene at once, the pixels in an event camera work individually and asynchronously. The pixels are triggered by changes in light, producing no data when no change occurs.

For example, a dark scene at night that contains a small light that blinks on and off. Using a 12MP, 30fps frame-based camera, the 360 million pixels worth of data would contain no useful information except for the area of pixels that contain the light source. Using an event camera in this situation however, only the pixels that detect the light source would send data to the user, producing millions of points of data less than the frame-based sensor.

This thesis studies ways to further improve the event camera performance from the perspective of the system user. Figure 1.1 shows a basic block diagram of the system.

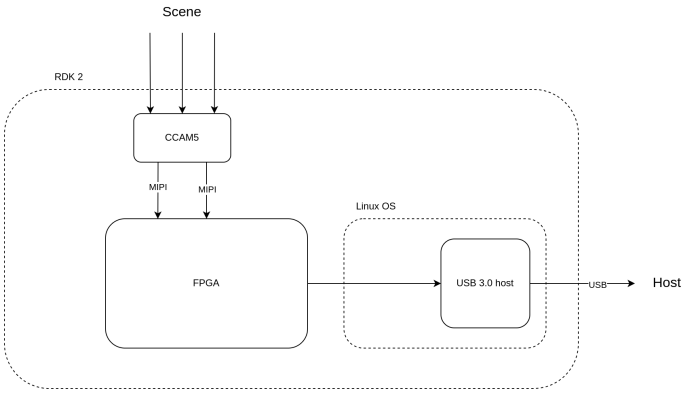

Figure 1.1: A basic view of the RDK2 system

The amount of pixels that a tracking algorithm requires to track a target is related to the relative speed between the target and the sensor. If the relative speed between a target and the camera is low, an object tracking algorithm does not need the entire resolution of the camera to track the target. As the sensing rate of the camera increases, the relative speed will decrease as it is related to the time span between consecutive images. Frame-based sensors generally have frame rates that are relatively slow, in the order of 30 to 60fps, unlike event-based sensors which have equivalent frame rates greater than 10 thousand frames per second [12].
This project uses *Prophesee's* Reference Design Kit (RDK) which is based on their Evaluation Kit 2 (EVK2) hardware platform with an IMX636 event-based sensor [5]. It is built around the *AMD Xilinx Zynq UltraScale+ MPSoC* Field Programmable Gate Array (FPGA). This kit uses the *CCAM5* module, which contains the IMX636 event-based sensor with a pixel resolution of 1280×720 and a maximum output rate of 1.1 billion events per second.

In this thesis I propose the use of a Region of Interest (ROI) filter to limit the number of pixels information is gathered from. I further suggest a method to maintain system performance while achieving this reduced data output.
Firstly, to establish a reliable computer system to enable this research, a robust work environment is discussed in Section 4. Secondly, the methodology of the experiments performed are detailed in Section 5. The proposed method is by using an ROI filter to limit information output from the FPGA, called events. I expect and show the relationship between the system performance from the perspective of the FPGA, to be closely related to the ROI in Section 6. These findings are related to the overall system latency in Section 7, before alternative methods to improve the system are given in Section 7.1.

1.1 Thesis contributions

The main contributions of this thesis are:

- A systems engineering trade-off analysis of virtualisation methods for the project work environment. Given in Section 4.
- An implementation of an FPGA ROI filter, Section 3.3.
- An analysis of the relationship between the ROI and the processing performance of the FPGA. Given in Section 6.
- A proposed solution to the negative time performance gain of an ROI filter in an event camera system. Given in Section 7.1.1.
- Two proposed alternative methods to an ROI filter for improving performance in an event camera system. Given in Section 7.1.2.

Chapter 2

Literature Review

This thesis centres around the processing performance of event cameras when used with Field Programmable Gate Arrays, in the aim of improving object tracking performance. Each of these three areas have been extensively researched, with little research performed on the combination of all three. This thesis aims to bridge this gap and study the use of all three technologies, tying these well researched fields together. The current literature for Object Tracking, Event Cameras, Field Programmable Gate Arrays (FPGA's) and their intersection is summarised in Sections 2.1, 2.2, 2.3 and 2.4 respectively.

2.1 Object tracking

Object tracking has been extensively researched for decades but is a complex field that still has many issues. Tracking continues to struggle with noisy images, complex object motion, non-rigid objects, complex shaped objects, changes in the illumination of the scene, and real-time processing [13]. For this thesis processing in real-time is directly relevant, however, the scenes' illumination is very relevant to the field of event cameras.

Some trackers combat illumination variation by using correlation filters such as Discriminative correlation filters (DCFs). Chen et al. [14] finds that correlation filter-based-trackers (CFTs) improve robustness, speed and the accuracy of the tracker in scenes of variable illumination. Huang et al. [15] showed that an "aberrance repressed correlation filter (ARCF)" improves robustness and accuracy when tracking objects in scenes with a lot of background noise, occlusion and illumination variation. This method was verified with footage recorded at only 10 frames per second (fps), however Bolme et al. [16] proposed a correlation filter based on squared error that exceeded 600fps using only grey scale images. If an algorithm such as this were applied to an event camera this is not an issue as event cameras currently do not support coloured output.

Other proposed methods to counter illumination variation include histogram equalisation, demonstrated by Sun et al. [17] who use a Bas-relief technique to recover any

information possible from an image. Li et al. [18] proposed another method that uses the shape, texture and colour of an image to create robust tracking in variably lit scenes. However, this method was shown to have limited real-time processing ability as it was presented. Their algorithm could only support real-time processing at 15fps for images with a pixel resolution of 160×120, or 3fps at a resolution of 320×240. While the resolutions may not be an issue in event camera object tracking through the use of a Region of Interest (ROI) filter as discussed in this thesis, the frame rate indicates that the processing speed of this algorithm is not optimal.

2.2 Event Cameras

Event cameras are a prominent alternative technology to frame-based sensors which are currently the standard. The applications of event cameras largely centre around tracking due to the equivalent frame rate speeds these cameras boast. The speed and low relative power consumption make event cameras ideal for real-time embedded systems. As this technology is still relatively new, there are a number of differences to standard frame-based cameras that need to be addressed to make these cameras viable. These issues include the difference in space-time output [19], and the difference in photometric sensing. In this context, the difference in photometric sensing refers to how standard frame-based cameras often operate in colour images while event-based cameras do not. Even when frame-based cameras generate grey scale images, they are fundamentally different from the polarity event cameras use in colour's stead.

Qinyi et al. [20] propose the use of a neural network architecture called *PointNet* that treats the event stream as three-dimensional point clouds for accurate real-time gesture recognition. They found that this method has one of the best accuracies of existing gesture recognition methods with an accuracy of 97.08%. However, this method reduces the number of input events down to only 40,000 events per second, which they state shows that it is viable for "real-time applications with strict timing and memory requirements". Due to the limited number of events in this method, it may not be viable for larger systems.
Cian et al. [21] use a *Prophesee* event camera, similar to what is used in this thesis, for a blink detection algorithm. This paper uses the entire pixel resolution of the camera, 1280×720 pixels, and shows that their algorithm has a wider range of accuracy than Qinyi et al., 98% at best and 80.3% at worst. They state that this is due to movement in the head, as their method sampled heads in static positions. The algorithm is proposed as real-time, however they state that groups of 50,000 events are accumulated over five milliseconds before any inference occurs. This alone introduces what is possibly too much latency in a true real-time system.

Ridwan [22] proposes a looming object detection algorithm that processes closer to the temporal limit of the event camera used, in the order of single microseconds. This algorithm is viable for real-time systems as it processes events individually rather than in batches such as the blink detection algorithm from Cian et al. [21].

2.3 Field Programmable Gate Arrays

FPGA's are popular mainly due to their flexibility and very high computational throughput. They also benefit from low power consumption, making them ideal alternatives to traditional Integrated Circuits (ICs).

Neural networks such as that proposed by Liu et al. [23] show the complexity and size of networks that FPGA's can handle while maintaining low power consumption. The network proposed here is able to simulate almost 17 million synapse connections with a power consumption of only half a Watt. This paper also proposed networks that were able to implement algorithms that recognise MNIST data with accuracies ranging from 83.16% to 97.81% from video with frame rates of nearly 210fps. This is in contrast to Saddik et al. [24] who showed that their architecture could only process up to 107fps, which they state "meets real-time constraints". For the FPGA System on Chip (SoC) used, Saddik et al. are still able to boast a maximum power consumption of only 1.8W [25], while a typical CPU power consumption is in the order of 50W [26]. Lamoureux et al. [27] found that energy efficiency can be improved by a further 14.6% by implementing efficient clock networks, as they found that the applications FPGA's are typically used in implement complex generic clock networks. This implies that if the system does not implement any clock networks, the energy efficiency would improve further again, potentially crucial in real-time embedded systems.

2.4 Object tracking using Field Programmable Gate Array's with Event-based Cameras

A real-time embedded object tracking system would benefit from the use of both event cameras and FPGA's as shown in the previous sections. Ordinarily, object tracking algorithms tend to use frame-based cameras as they are readily available, cheaper, and the standard for camera technology currently. The combination of FPGA and event camera technology is gaining popularity, but a majority of research does not have power or timing constraints strict enough to warrant the complexity of using these technologies together. However, there does exist some research utilising all of these technologies.

Yizhao et al. [28] present a reconfigurable FPGA architecture which performs real-time multi-object tracking from event data. They found that their architecture processed almost 3 million events per second with a maximum power consumption of 5.45W. Comparing this to a baseline software trial, Yizhao et al. found that their architecture had a throughput 44 times higher and a power efficiency 35.4 times higher. They also found that their worst-case latency was almost 100 times less than the baseline.

Barrios-Avilés et al. [29] suggest that the cause for event camera technology not being a standard in industrial applications despite their benefits is due to the technology not yet having established data buses for transfer. As it stands event systems use much higher bandwidth than most industrial systems can handle, with the simple fact that an event camera generates so many events it floods data networks. Barrios-Avilés et al. create a network node based on an FPGA that is able to handle this data accordingly. They test this by using a two axis robot that uses event cameras for fast image recognition, something that would be used in industrial applications. They were able to reduce event data by up to 85% while achieving a 99ms increase in position detection compared to the traditional frame-based camera.

Another paper by Barrios-Avilés et al. [30] further investigate the reduction of event data by implementing a filtering algorithm to reduce data without information loss. They verify their filter by implementing an object tracking algorithm and process the data on a *Xilinx Virtex6* FPGA. They found that their filter decreases error compared to the baseline filter, simultaneously reducing data by 85%. From their results, Barrios-Avilés et al. conclude that a system that implements an event data reduction filter such as theirs, processed on an FPGA, shows "real time operation capabilities and very low resource usage".

Chapter 3

Background

This thesis requires knowledge on five main topics, namely: event cameras, latency, Field Programmable Gate Array's (FPGA's), the Yocto project, and robust work environments. This will be discussed in Sections 3.1, 3.2, 3.4, 3.5 and 3.6 respectively.

3.1 Event Cameras

To understand how an event-based vision system works, an understanding of how a frame-based vision system works is needed.
Standard cameras, such as those used in Smartphones and Security Cameras, are what are called frame-based cameras. The camera's resolution comes from the size of its pixel array, a grid of rows and columns of individual pixels. These cameras work by capturing an image as a shutter exposes the pixel array to a scene. A shutter can either be something that mechanically covers the pixel array or logic that electronically turns off the pixels. The output from this exposure is called a frame, and the rate frames are captured is called the frame rate.

The time between exposures of the camera defines the frame rate of the camera. To sample the image, the camera reads the intensity of light at each pixel in the camera's resolution. In a situation where the scene does not change much between consecutive frames, such as at night, unnecessary data is captured and processed.
An event-based camera does not use frames or shutters to sample pixels at a fixed interval. Instead, the pixels of an event-based sensor are triggered by a change in light intensity and are independent of one another. This difference results in event cameras having much higher frame rate equivalents compared to frame-based cameras, with the difference between technologies illustrated in Figure 3.1.

When a pixel captures a change in brightness, it creates what is called an event, which contains data on the position of the pixel, the timestamp of when the change happened, and the polarity of the brightness change. The event data format is illustrated in Figure 3.2.

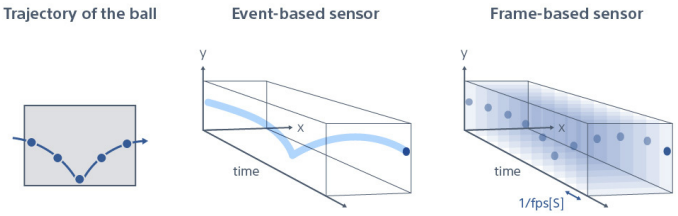

Figure 3.1: Comparison of how a Frame-based sensor views an action to an Event-based sensor [1]

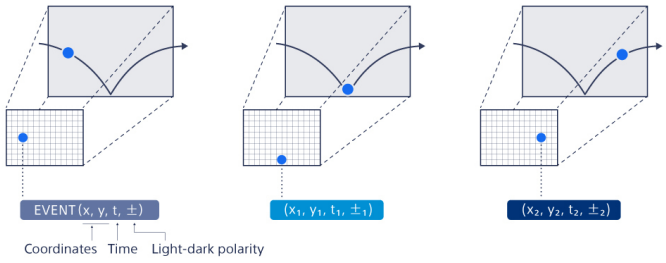

Figure 3.2: Format of events [2]

The typical notation used to describe an event is given as [31]:

$$e = [x, y, t, p]^T$$

Where **e** represents the event, and **x** and **y** represent the x and y coordinates of the pixel respectively. **t** is the timestamp, denoting when the event happened to a microsecond resolution [1]. **p** represents the polarity, given as a binary value showing whether the change in brightness was either an increase or decrease.

When an event is registered by a pixel, the pixel will create an event in this form, and send an alert to an arbiter [32]. The arbiter then sorts through the pixel array by determining the priority of the requested events based on the row coordinates [33]. The arbiter reads through rows of pixels, skipping rows if no request has that row coordinate. It then adds requested events it finds to a buffer that holds events to get output. This method of searching is much faster than going through every pixel and determining if there is an event there as it is able to potentially skip numerous pixels. For an $n \times m$ camera pixel resolution, $(width \times height)$, a linear search would take $O(n*m)$ time, while a row selection algorithm would take only $O(n+m)$.

The event camera used in this project is the *CCAM5* module with the *IMX636* event-based vision sensor [34]. It has a pixel resolution of 1280×720 and is able to output a maximum of 1.1 billion events per second as used in this project. It connects to the rest of the system used in this project with two MIPI CSI-2 lanes capable of transmitting 1.5 billion bits per second per lane.

3.2 Latency

This project focusses on the performance metric of latency. Latency in an embedded system is the "time delay between input event being applied to a system and the associated output action from the system" [35]. In simpler terms, latency is the time it takes a process from start to finish, and jitter is a measure of the consistency of the latency.
For the RDK used in this project, latency is "*the pixel response delay to a temporal contrast step*" [3]. This definition is generic and relative to an axis of time. This project requires an understanding of latency from different perspectives, namely the camera, the FPGA, and the user.

The perspective of the camera is best illustrated with Figure 3.3 which has a time axis that shows when a lighting change occurs and when a pixel in the camera detects this change. This figure also shows jitter, another important performance metric in embedded systems that, for the RDK, "*corresponds to the temporal precision at pixel level*" [3].

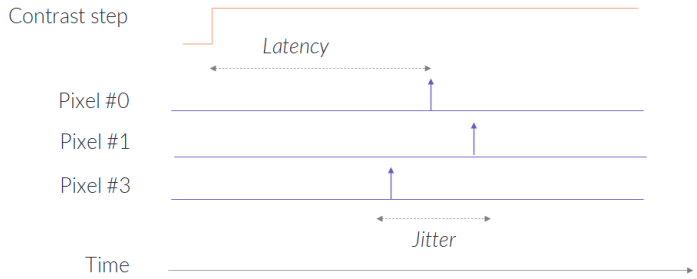

Figure 3.3: Visualisation of latency and jitter on time axis in reference to the event camera used [3]

From the perspective of the FPGA, latency is of primary focus in the experiments explained in Chapter 5 and measured in Chapter 6. This is shown as $\Delta\hat{t}$ and ΔT in Figure 3.4, which shows three different time axes representing perspectives within the RDK2 system shown in Figure 1.1. In real-time, there are four time axes which includes the addition of the real-time or physical time axis. For the event camera system,

this represents when a change in light takes place. For simplicity, this axis is combined with the camera axis, which represents when the event camera captures this lighting change and creates an event. They were combined as the timing resolution of the event camera used in the RDK is very small, less than a microsecond, and as such for these purposes is negligible and out of the project scope.
The other two time axes are the FPGA and user time axes. Respectively, these represent when the FPGA registers an event after any transfer delay from the camera to the FPGA, and when the FPGA registers an event from the perspective of the user. The time axis from the perspective of the user, includes all previous delays described, along with delays that come from the transfer from the FPGA to the user. This primarily includes delay from the USB driver that connects the user to the RDK system.

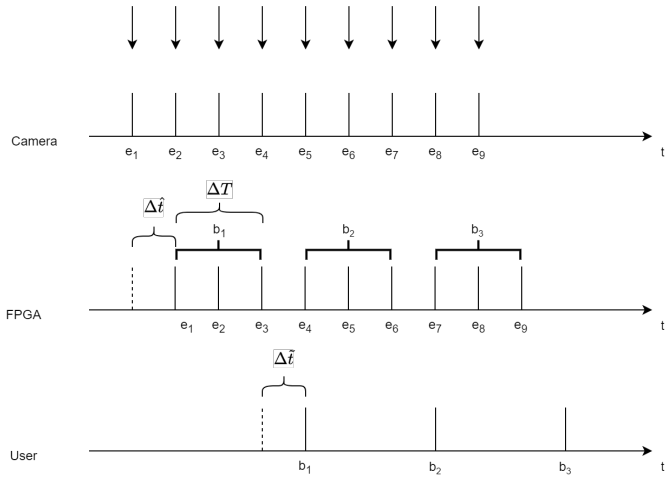

Figure 3.4: The different axes of time relative to each other

The total latencies in reference to the FPGA and the user is described in Equations 3.1 and 3.2 as seen in Figure 3.4.

$$\zeta_{FPGA} = \Delta\hat{t} \tag{3.1}$$

$$\zeta_{user} = \zeta_{FPGA} + \Delta T + \Delta\tilde{t} \tag{3.2}$$

Where ζ is latency and the subscript shows the perspective. $\Delta\hat{t}$, ΔT and $\Delta\tilde{t}$ are time variables, illustrated in Figure 3.4.
$\Delta\hat{t}$ is the delay between the camera and the FPGA, ΔT is the difference in timestamps between the first event in a buffer to the last event. $\Delta\tilde{t}$ is the delay from after a buffer is sent from the FPGA to when the user receives it.

For the scope of this project, I focus on ΔT and $\Delta\tilde{t}$. ΔT represents the ΔT_{buffer} described in Section 6.1.1.4, and is related to all other measurements described in Section 6.1.1. From the scope of this project, from the perspective of the user the

performance of the system is only concerned with ΔT. As the goal of this project is to increase the performance of the RDK, I show the effect of reducing the number of events on metrics such as ΔT, and relate it back to the overall $\Delta \tilde{t}$.

3.3 Region of Interest filter

In object tracking the tracking algorithm only needs a select area around the object being tracked, any excess information is not needed. An ROI filter specifies an area and excludes all events outside this and is visualised in Figure 3.5.

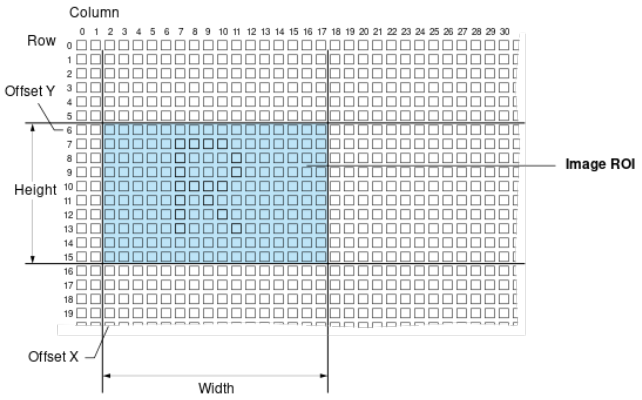

Figure 3.5: A Region of Interest (blue) in a pixel array [4]

Figure 3.6 shows a screen capture of the output event stream from the RDK without any filters, showing the full pixel resolution of the CCAM5. Events are represented as blue and white pixels and are seen covering the full 1280 × 720 area coming from uniform noise. The output shown in figure 3.7 comes from the RDK with an ROI filter applied to the FPGA, restricting the area down from 1280 × 720 to 150 × 150. The event streams shown in these figures were captured in the same environment, with the same lighting conditions resulting in very similar noise. The area of interest in Figure 3.7 was the upper left corner of the image, representing a square with the coordinates (0,0) and (150,150) in the CCAM5's coordinate system. This figure shows the absence of events in the rest of the frame.

An example of where this filter would be useful in a tracking algorithm would be tracking a torch moving in the dark. An object tracking algorithm would not need to process any pixel representing the background, as it only needs the pixels around the area of the torch. The goal would be to feed updated areas of interest from the tracking algorithm into the event-based camera, moving the ROI and tracking more efficiently with reduced processing.

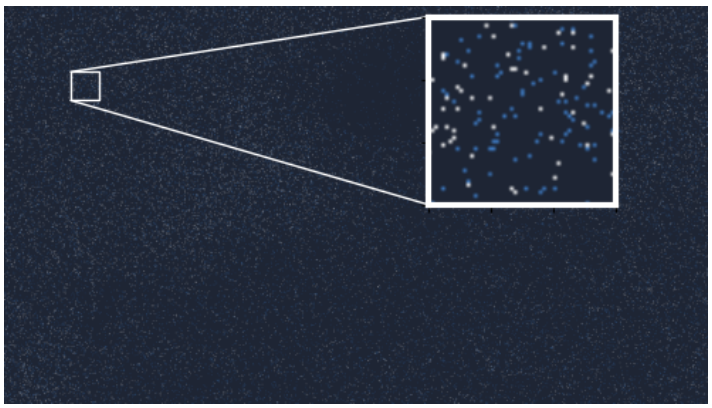

Figure 3.6: Visualisation of the RDK camera output with no filters applied

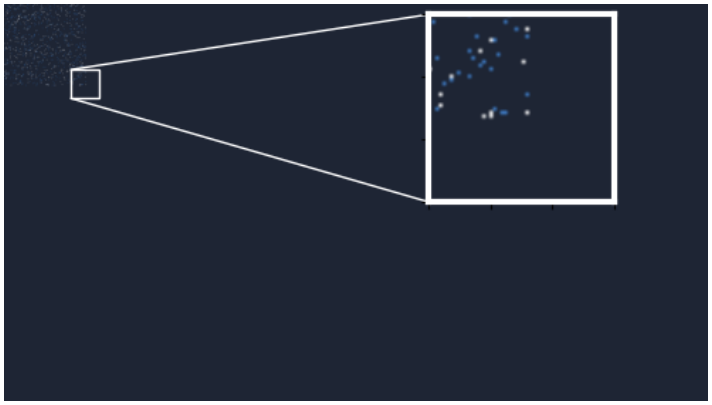

Figure 3.7: Visualisation of the RDK camera output with a 150 × 150 region of interest filter applied

3.4 Field Programmable Gate Array (FPGA)

An FPGA is a type of Integrated Circuit (IC), which is built so that the user can reconfigure it. ICs are small silicon wafers containing components such as transistors, capacitors and resistors [36] etched into the material. Conductive materials, commonly copper, are placed between the components to connect them. In normal ICs, these paths are created during the manufacturing process and cannot change as they are made with a specific purpose in mind.
In FPGA's, components are not etched into the semiconductor. Instead, the FPGA contains smaller logic blocks, which in turn contains flip-flops, lookup tables and adders. The FPGA is configured through a Hardware Description Language (HDL), which produces a bitstream. A bitstream is a binary file that contains the configuration and is loaded into the FPGA at boot. One such creation method of *HDL's* is *Xilinx's Vivado* [37], which is a software suite for *HDL* synthesis and design.

The FPGA in *Prophesee's RDK* uses a bitstream designed in *Vivado*. *Vivado* provides

a visual interface to design and create the bitstream for an FPGA. Bitstreams define the hardware configuration of FPGA's, and *Vivado* translates input, such as seen in Figure 3.8, down to the level of describing where a logic gate should be. The compilation chain required is immense, relying on numerous packages to control different parts. This is discussed further in Section 3.6.

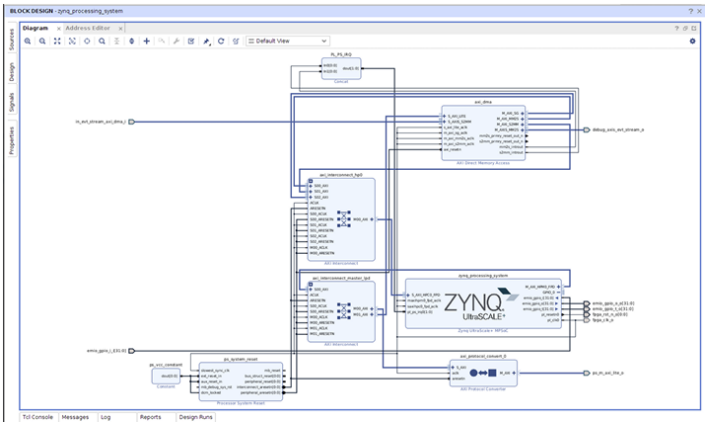

Figure 3.8: Vivado block diagram for this project [5]

The FPGA used in this project is *Xilinx's Zynq UltraScale+ XCZU6EG-FFVC900-1-E MPSoC* implemented on a module made by *Trenz*. This FPGA has an Application Processor Unit (APU) built on a quad-core ARM Cortex-A53 processor. It has more than 450 thousand logic cells and 256KB of internal memory. The FPGA interfaces with "*PCIe Gen2, USB 3.0, SATA 3.1, DisplayPort and Gigabit Ethernet*" [5].

3.5 Embedded Linux with the Yocto Project and PetaLinux

Linux is successful largely due to its open source nature, and due to it being more lightweight than other operating systems, enabling it to run on most systems. It is common for developers to redistribute Linux for different systems by creating a new distribution. As Linux is open source, it is possible to alter a distribution to only include the features and packages it requires.

The Yocto Project, or Yocto for short, is an open source project for Linux distribution development. Yocto makes it simple to redistribute a custom Linux distribution and is "*the foundation of a whole sector of embedded Linux, as well as being a complete build system*" [38].

PetaLinux [39] is an abstraction of Yocto combined with *Xilinx* specific layers and extra tools [40]. *PetaLinux* was created by *Xilinx's* parent company *AMD*, and is con-

sidered as Yocto for *Xilinx* devices.

This project uses two files that are output by *PetaLinux*: *BOOT.BIN* and *image.ub*. *BOOT.BIN* is the boot binary image that contains the bitstream as well as some *Xilinx* specific configuration such as a Platform Loader and Manager (PLM), and general bootloader binaries such as the universal bootloader (u-boot) and a first stage bootloader (FSBL) [41]. *image.ub* is the Linux kernel image in u-boot format. During the experiments described in Section 3.3, this project only changes the bitstream, and as such only the *BOOT.BIN* needs to be updated.

As *Prophesee's RDK* is built around *Xilinx's Zynq UltraScale+ MPSoC* FPGA, *Prophesee* used *PetaLinux* to develop its custom *Linux* distribution. This *Linux* image is written to the QSPI flash memory of the *RDK*. The image contains the bitstream used to configure the FPGA, as well as all the mechanisms and protocols needed to make the system usable. One such system is the USB protocol, which is used to transmit the data that comes from the *CCAM5*, through the FPGA, out through a connected USB to a host computer.

3.6 Robust work environment

In the context of this thesis, a work environment refers to the setup of the computer used throughout the project. The project primarily involves work on a computer. This project uses specific software, namely *Vivado* and *PetaLinux*, discussed in Sections 3.4 and 3.5 respectively.
A robust environment refers to an environment that stays constant, is predictable, and is reliable. This project needs a robust work environment to prevent software issues. In software development, most issues come from '*Dependency Hell*' [42], which occurs from numerous conflicting dependencies installed. As software advances, it becomes more complex, increasing in libraries and packages called dependencies. Two different versions of a single package cannot be installed to the same system. In large systems with hundreds of dependencies across multiple projects, managing these dependencies and their versions becomes tedious, difficult, and error-prone.
In order to avoid this problem, packages need to be kept separate so that they don't conflict. The simplest solution is to use versions of programs that don't conflict with each other, however this project's constraints include the use of specific versions of software. While one solution could be to use different computers for conflicting software, this is not a realistic solution for many, including this project.
The emulation of computers on a single host is called *Virtualisation* [9], in which there are two primary methods, via virtual machines [11], discussed in Section 4.2, and via

containers [8] (also called *containerisation*), discussed in Section 4.1. The work environment used in this project is discussed in Section 4.3.

The experiments performed in this project are explained in Section 5, with the explanation and discussion of the measured results in Section 6. This thesis concludes its findings in Section 7 before detailing further work on this topic in Section 7.1.

Work Environment Design

Virtualisation was investigated to provide protection against potential software compatibility issues that may come later in the project. Containers and virtual machines were analysed early on in the project, in the hope that if implemented I could be confident that my applications, namely *Vivado* and *PetaLinux*, will work as intended. The project was set with realistic expectations, knowing that the ultimate goal of the project was not to design a work environment. Unless a viable virtualisation solution was found, I would use native installations, and assume the risk of future issues.

While Virtual Machines and Containers are different technologies that accomplish different tasks, they are both virtualisation technologies meaning that they share similarities in their high level goal, simulating a workspace. I will look at the strengths and weaknesses of the two main methods of virtualisation, containerisation and via virtual machines in Sections 4.1 and 4.2 respectively. The final decision based on these sections is discussed in Section 4.3, and the success of this decision is shown in Section 4.4.

4.1 Containers

Containers are designed to simulate a computer's operating system. Containers are runtime environments, which are sub-systems within the computer, and are made to run on-top of a computers operating system, not touching the hardware. A container takes advantage of this by sharing the operating system with the physical native host, resulting in a lightweight environment. This means that a user isn't able to use or experience a different system. An example of this would be that on a native Windows host, a container would not be able to truly run Linux. In this example, Windows could use Windows Subsystem for Linux (WSL) to emulate a Linux kernel, but that is all it is, an emulation, as any WSL functions are translated to functions that Windows can understand. A container can emulate the Linux kernel, but it cannot emulate a Linux operating system. Despite this, a container provides independence and separation of programs and applications, making it simpler to run multiple, potentially conflicting, applications in parallel.

A container can be viewed as a snapshot as it doesn't automatically update on its own. The state it was created in will be the state it stays in. This may be seen as a negative if a container environment was used as the main development environment, but generally containers are used in situations where this is a positive. Projects that require specific dependency versions would benefit from this as there's no risk that it will update and render the project unable to compile. In production this is especially desirable as developers would want as much control as they can get, and not be susceptible to their products suddenly not working because a package was updated and is incompatible with a different package.
When container environments are installed, often the user will configure a limit to the system's resources that the containers can use. This is a limit and does not actually allocate the resources to the container, but results in a fluid use of memory between the native host and the containers. If the containers are complex or there are a lot of them, this can cause some bottlenecks as they reach this limit however, causing the containers to fight for the resources.

Table 4.1 shows the strengths and weaknesses, as just discussed, in a table format for ease. This table and Table 4.2 were used to design the work environment used in this thesis.

Table 4.1: Strengths and weaknesses of containers [6] [7] [8].

Containerisation	
Strengths	**Weaknesses**
Lightweight and portable	Can't run different operating systems to the host
Doesn't update on its own	Requires destroying the container to update
Reduces hardware cost through consolidation	Shares resources for applications within the same container

4.2 Virtual Machines

The goal of virtual machines is to simulate a system down to the hardware level and be as independent of the native physical system as possible. To achieve this, virtual machines require their own operating systems and a software program called a hypervisor to communicate between operating systems. This fact makes virtual machines more resource intensive and less lightweight than a container. Along with this, as a virtual machine simulates the hardware of a physical machine, the physical system's resources;

CPU, memory, and RAM, are split between the physical and virtual machines. When a virtual machine is first created, the installer partitions system resources, assigning them to the physical machine and the virtual one. This means that the resources allocated to the virtual machine are entirely its own, nothing outside the virtual machine will use them, but they are shared by processes within the virtual machine. As a virtual machine contains so much and is woven so tightly into the physical hardware, it means that it isn't realistic to move virtual machines between hosts or share them with colleagues. A brand new virtual machine would need to be installed on a different system.
These costs do not come without benefits. By having a separate operating system, you are able to use multiple operating systems on one physical computer at the same time. This is especially useful for projects that may have applications that only run on specific operating systems, but the team is constrained to a single host. Junior developers or researchers at the start of their project would benefit from being able to experiment and find the system that best fits easily. Another benefit to virtual machines is that because they are a fully enclosed system, they get the advantage of supported updates. Just like any other computer, a virtual machine could be configured to automatically update, resulting in very low maintenance for the user.

Table 4.2 shows the strengths and weaknesses, as just discussed, in a table format for ease. This table and Table 4.1 were used to design the work environment used in this thesis.

Table 4.2: Strengths and weaknesses of virtual machines [9] [10] [11].

Virtual Machines	
Strengths	**Weaknesses**
Can run a different operating system to the host	Are quite resource intensive and not portable
Can be updated without needing to be reinstalled	Susceptible to automatic updates
Reduces hardware cost through consolidation	Shares resources for applications within the same virtual machine

4.3 Decision

As discussed in Section 3.6, this project heavily relies on two *Xilinx* applications, namely *Vivado* and *PetaLinux*. These applications are both extremely complex and layered in terms of dependencies.

In Section 3.6 I discuss the concept of '*Dependency Hell*', and for a complex system such as *Vivado* or *PetaLinux*, this is a situation to be avoided at all costs. Working versions of these applications should be viewed as fragile, as simply having a conflicting package installed elsewhere on the computer can cause them to no longer work. While *Vivado* and *PetaLinux* are separate applications with different features and goals, they still share dependencies, and it would only take one conflict to cause major issues. For this reason virtualisation was researched as a possible solution, as my project was constrained to one computer, meaning that physically separating the programs was not an option. While both containerisation and virtual machines are excellent technologies, they are only possible solutions, especially as creating a robust work environment was not the end-goal of this thesis.

After detailed comparisons of the two technologies and designing implementations, it was decided that there is no viable virtualisation for this project. I identified that containers met my requirements better than virtual machines and so pursued them. As the applications needed for my project were so complex, they had intense resource consumption, meaning that the overhead of virtual machines made them a poor candidate. For this project, I recommend using at least 16 GB of RAM and an SSD storage of at least 1 TB. Additionally, the way virtual machines handle updating, while it could be managed, made it less than ideal for my project. Consequently, I decided that out of containerisation and virtual machines, a container would best fit my requirements; not being too resource hungry and providing an isolated version snapshot with no risk of change.

Researching and designing containers, I found that while in idea a container suited my needs well, in application they were difficult to create. Containers had various issues such as directing the Graphical User Interface (GUI) to the host not the container, and the issue of USB pass-through between the host and the container. I believe that it is possible to design a container for *Vivado* and *PetaLinux*, but given my position I decided that the difficulty of designing it was out of scope for the project. As such I found natively installing my programs to be the best solution, despite the risks.

4.4 Result

After installing both *Vivado* and *PetaLinux* I recompiled the respective projects provided by *Prophesee*, verifying that the installations were a success. In order to make changes to the RDK, I needed to be able to re-flash, or update, the currently running Linux image and FPGA bitstream. *Prophesee* provided two methods to do this in their Technical Reference Manual [5], namely the *fastboot* protocol and directly overwriting

sections in memory using the *dd* command. For memory safety and protection against accidentally writing data to the wrong area in memory and overwritting other parts, I decided that using the *fastboot* protocol was the best option. In order to verify that the updates worked, I made changes to both the *PetaLinux* image and the *Vivado* bitstream. The change to the *PetaLinux* image was nothing of note, simply changing text in a file, but the bitstream was not as simple to verify that it had changed. As my best option was to make a visual change, I decided to flip the polarity of all the events coming from the camera.

The changes were successful and the results of the polarity flip can be seen in Figures 4.1 and 4.2. The experimental setup needed to be reproducible and as consistent as possible. For this a simple stage was created as seen in Figures A.1 and A.2 in Appendix A, with the RDK placed as shown in Figure A.4 in Appendix A. The RDK was moved along the edge of the stage directed at the black and white background as can be seen in Figure A.3 in Appendix A. The stage was designed to allow light to filter through the top and from behind the camera, while limiting shadows from the sides. The camera was moved methodically from one side to the other, and the camera would detect the edges of the black and white background moving as events. This resulted in an event stream of two clear lines of events, one showing the light decreasing (a negative event) as the black moved into frame, and one showing the increase in light (a positive event) as the black moved out of frame.

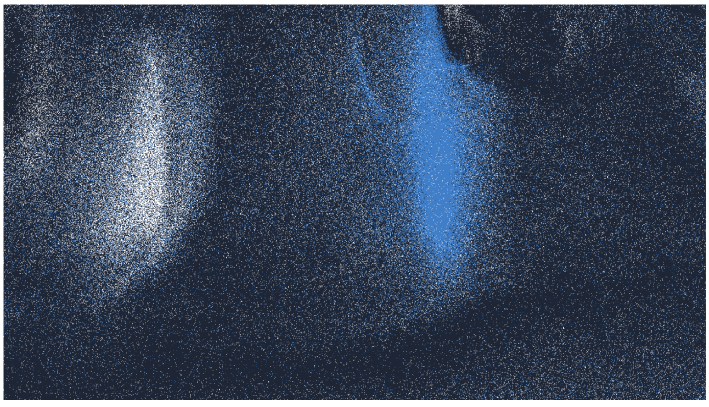

Figure 4.1: Left to right movement of the camera in the stage with the default event polarity

For both experiments, the camera was moved from image left to image right. In Figure 4.1 the default polarity has the blue pixels denoting negative events and white pixels for positive events. Once the polarity was flipped, the roles of the two coloured pixels reverse, with blue pixels now showing a positive event and white pixels showing a negative event, as seen in Figure 4.2.

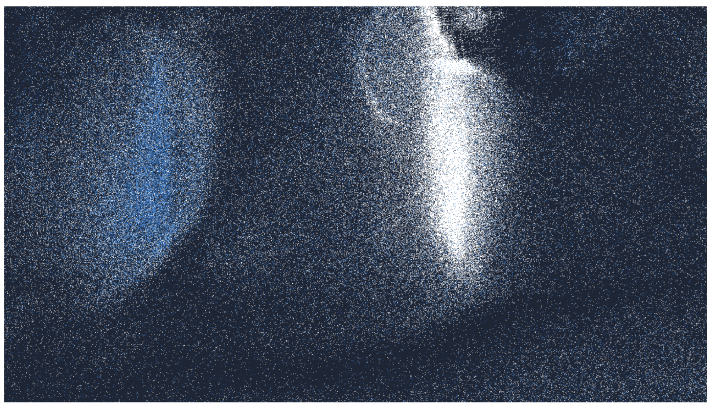

Figure 4.2: Left to right movement of the camera in the stage with the flipped event polarity

Chapter 5

Methodology

The objective of this thesis is to investigate ways to increase the processing speed within the FPGA of the *Prophesee* RDK. The *IMX636* event-based sensor the RDK's CCAM5 module uses has a resolution of 1280 × 720 pixels and can output up to 1.1 billion events every second [34]. To maximise the benefit from the sensor's speed, any processing and transmission that occurs between the sensor and the user must be as fast as possible. This chain of processes is complex, as such this project focuses on one link in this chain, the processing within the FPGA.

The CCAM5 outputs events directly into the FPGA, as shown in Figure B.1 in Appendix B. As the FPGA is the first sub-system in the chain, after the CCAM5 itself and the connector between the two, any improvements found will likely have a flow on effect to the subsequent sub-systems. This fact makes the FPGA an ideal target for this study.
Figure B.1 in Appendix B shows the overall structure of the FPGA, and it is clear that this system is complex which means it has multiple places a possible bottleneck could occur. Figure B.2 in Appendix B shows the structure of *huahine Interface*, the FPGA sub-system largely responsible for processing the event data. I identified this as the most likely place for a bottleneck to occur, and so focused my effort there.

This projects' goal is to increase processing speed for tracking applications. By filtering the input event stream to discard events outside a certain area, I aim to show the effects this has on processing speeds and data transmission.

Section 3.3 explains the Region of Interest (ROI) filter used in the experiments described in Section 5.1.

5.1 Experiment

In order to quantify the effect an ROI filter had on data processing, I performed ten experiments which each consisted of 100 runs. Firstly I collected baseline data, using a bitstream that had no filters applied, then I performed experiments with ten different

ROI filters. Each filter had the same aspect ratio as the full resolution of the camera, but each at different percentages of the height and width. For example, one experiment filtered the resolution to 10% of the full resolution, representing a pixel resolution of 128 pixels wide by 72 pixels high. The ten experiments ranged from 10% to 100% in 10% increments. An experiment with a filter with the full 100% resolution was performed to view the effect the filter alone has, before any events are filtered. Each experiment consisted of 100 runs to provide more accurate measurements and reduce the effects of outliers. Each run lasted 45,000 microseconds, calculated from the timestamps of the events, to avoid any outside influence that could come from timing through the Real Time Clock (RTC) of the computer. The RDK was set up within the stage as shown in Figure A.1, with the camera pointing toward a corner at a distance of about an inch and a half, shown in Figure A.5 in Appendix A. A C++ script, described in Section C.1 in Appendix C was made that used API's from *Prophesee* to initialise an instance of the CCAM5 to begin streaming the data. As the data was streamed, the events were captured in a Comma-Separated Values (csv) file containing the event timestamps and a flag describing the events' relation to the buffer it came in. The script was designed to reinitialise the camera instance for each run, to remove any chance of overflowed buffers from previous runs. Each run wrote to a separate file for ease of access. Once 100 runs were collected from the RDK with no filters, the bitstream was updated using *fastboot* as described in Section 4.4 to contain a ROI filter. The next experiment was then started, re-running the C++ script again. The RDK stayed in the same position between experiments which were performed within minutes of each other.

A Python script was created, described in Section C.2 to analyse this data and calculate different measurements, which will be presented in Section 6.1. Measurements such as the average time between consecutive events and buffers were calculated as they quantified the effect of the ROI filter, showing its effects on the systems delay. This script also produced graphs depicting various patterns of interest, such as the spread of output events and event buffers through time. These graphs will also be shown and analysed in Section 6.1.

In creation of this report, two more *python* scripts were created. Script C.3 generates the plots Figures 6.1, 6.2, 6.14, 6.7, 6.15, and 6.18, as well as Tables 6.1, 6.2, D.2, and D.1. Script C.4 alters images to magnify certain areas and was used to generate Figures 3.6, and 3.7.

Chapter 6

Results and Analysis

6.1 Region of Interest

This section will discuss the results from the Region of Interest (ROI) experiments detailed in Section 5.1. Table 6.1 summarises the findings from the experiments that will be discussed in the subsequent sections, and Table 6.2 gives ratios comparing each measurement in each experiment to the respective measurement in the baseline experiment. The ROI experiments measured different aspects of the output event stream buffer from the RDK such as the time between consecutive events and buffers, and the rate at which the buffer filled with events. Taking the results from the ten experiments, graphs were constructed and provide clear visual descriptions the change in ROI has on FPGA processing.

From these results it is clear that decreasing the ROI increases the delay on the level of individual events. The ROI algorithm is built into the bitstream, and as such does not affect the CCAM5 module. This means that from the perspective of the camera the same number of pixels is monitored by the arbiter, regardless of the ROI. Within the bitstream, the ROI filter ignores events coming from outside the ROI coordinates, simply discarding them and not passing them on to the USB buffer to be sent to the client. Due to this fact, as the event camera arbiter still needs to go through the full pixel resolution of the camera. While the CCAM5 has a time resolution of one microsecond, it still takes the arbiter time to collect events from individual pixels. There is a finite amount of time between when a light change is detected by a pixel and when the camera arbiter assigns that event a timestamp, and in scenes with a lot of light which cause a lot of events this is exaggerated.

As the ROI filter is integrated at the FPGA level and the camera sensor is unaffected, it is expected that decreasing the ROI will increase the time between events as a function of the pixelrow height of the ROI. This is due to the way the row based algorithm the arbiter takes to iterate through the pixel array, as it assigns the same timestamp to each event in the same row [43]. When the ROI decreases and the pixel array dimensions decrease, the time the arbiter spends assigning timestamps to pixels

outside the area becomes more noticeable. This leads to larger ΔT values, which is exactly what I see as shown in Sections 6.1.1.2, 6.1.1.3 and 6.1.1.4.
The relationship is expected to be given by the function shown in Equation 6.3, which describes the inverse relationship between the ROI pixel height and the measurement.

$$A = ROI_x * ROI_y \tag{6.1}$$

$$f(A) = \frac{A \times y_{100}}{10,000} \tag{6.2}$$

$$f(ROI_y) = \frac{100 \times y_{100}}{ROI_y} \tag{6.3}$$

Where ROI_y is the pixel height of the ROI as a percentage of the full sensor height. $f(ROI_y)$ represents a measurement explained in Sections 6.1.1.2, 6.1.1.3 or 6.1.1.4. y_{100} is the measured value at an ROI 100% of the full sensor dimensions. The value of 100 in Equation 6.3 is used as the measured value at an ROI pixel height percentage of 100 was used to scale and anchor the curve.
The value of $10,000$ in Equation 6.2 is used as the measured value at an ROI area percentage of 100 was used to scale and anchor the curve. $10,000$ is the square of 100, used because the area is found with Equation 6.1.

6.1.1 Measurements

For each of the ten experiments performed, six measurements where calculated and analysed. These measurements are:

- Overall average buffer size across the experiment, measured in number of events;
- The time difference between consecutive events, ΔT_{event}, measured in the time resolution of the RDK given in μs;
- The length of time it takes for a buffer to fill with events, ΔT_{buffer}, measured as the time difference between the first event in the buffer and the last given in μs;
- The rate at which buffers fill with events, measured in number of events per μs;
- The time delay between consecutive event buffers, measured in the time resolution of the RDK given in μs; and
- The total number of events that were processed across the experiment, measured in number of events.

Table 6.1: Table of results for all ROI experiments

	Experiment										
Measurement	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**	**Baseline**
Buffer size (events)	316	317	317	317	317	317	316	318	318	318	318
ΔT_{event} (μ s)	1.96	0.89	0.6	0.47	0.41	0.34	0.29	0.25	0.23	0.21	0.21
ΔT_{buffer} (μ s)	611.78	280.82	189.98	150.11	128.9	109.44	91.91	79.51	71.76	65.92	67.98
Event rate (events/μ s)	2.19	2.4	2.63	2.88	3.14	3.45	3.81	4.2	4.56	4.9	4.75
Buffer delay (μ s)	2.13	0.9	0.6	0.47	0.41	0.35	0.29	0.25	0.23	0.21	0.21
Total number of events (events)	97189	106555	116956	128075	139549	153109	169231	186393	202100	217406	210599

Table 6.2: Table of results for each ROI experiments ratios relative to the baseline

	Experiment									
Measurement	**10%**	**20%**	**30%**	**40%**	**50%**	**60%**	**70%**	**80%**	**90%**	**100%**
Buffer size (%)	99.2	99.5	99.5	99.5	99.6	99.5	99.4	100.0	100.0	100.1
ΔT_{event} (%)	917.1	414.7	280.2	221.3	189.9	161.3	135.4	117.0	105.6	97.0
ΔT_{buffer} (%)	899.9	413.1	279.5	220.8	189.6	161.0	135.2	117.0	105.6	97.0
Event rate (%)	46.1	50.6	55.5	60.8	66.2	72.6	80.3	88.5	96.0	103.3
Buffer delay (%)	997.7	422.2	281.5	221.1	190.5	161.9	135.8	117.1	105.6	97.3
Total number of events (%)	46.1	50.6	55.5	60.8	66.3	72.7	80.4	88.5	96.0	103.2

The results in Table 6.2 come from the same data as Table 6.1. Each table shows the six key measurements taken from all ten experiments. All the measurements, except the total number of events, are the calculated average values across each experiment. During the analysis of each experiment, these five measurements were calculated for each buffer. The experiment average was then taken as the average values from all buffers. The total number of events was not an average but a count of events across the experiments. Table 6.2 show the relation between the values of the experiments and the baseline in Table 6.1. These relations were calculated as the ratio between the experiment and baseline values using Equation 6.4.

$$ratio = 100 * \frac{v_s}{v_b} \tag{6.4}$$

Where v_s is the experiment measurement value and v_b is the baseline measurement value.

Using the values in Table 6.1, plots were created for the measurements across the ten experiments. These plots are shown and discussed in Sections 6.1.1.2, 6.1.1.3, 6.1.1.4,

6.1.1.5 and 6.1.1.6. Each plot shows the expected behaviour and also contains a line of best fit approximating the direction of the data. The lines of best fit are Least Squares polynomial fit models, and the expected relationships are calculated using functions discussed in each respective section. One way to measure the correlation between a trend-line and the data is to find the Pearson correlation coefficient (r) values, which is a value in the range of $[-1, 1]$, where the further the value is from 0 the stronger the correlation. The r values for both the expected trends and lines of best fit are given in Table D.1.

For each measurement, Table D.1 shows that the expected relationships almost perfectly correlate to the data, proving that my hypothesis was correct.

The error of both trend-lines were calculated using the Root Mean Squared Error (RMSE) formula given in 6.6. RMSE measures the error of individual points in the approximation to the actual points in the data, which provides a measure of the accuracy of the models. The RMSE values calculated for each plot is shown in Table D.2, organised by data measurement.

$$\hat{x} = fit(y, x) \tag{6.5}$$

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \hat{x_i})^2}{N}} \tag{6.6}$$

Where N is the number of points in the model, x_i is the measured ith value of the data and $\hat{x_i}$ is the ith approximated value given from the regression algorithm represented as the function *fit(y, x)* in Equation 6.5.

6.1.1.1 Buffer size

The buffer size measurement shown in all three tables represents the number of events that are in each of the buffers transmitted from the RDK. The values shown in Table 6.1 for the buffer size are the average size of the buffers in each experiment. This is measured in the number of events and is always an integer. The values shown here are average values across all buffers rounded to integer values. Individual buffer sizes were calculated using Equation 6.7, and the average was then calculated using Equation 6.8 before being rounded.

The values shown in Table 6.1 shows some deviation, seen by the second order line of best fit in Figure 6.1. This figure shows us that the buffer size stays relatively constant. The expected trend has an RMSE value of 0.645 which is very small relative to the scale of the data, showing the close fit of the approximation.

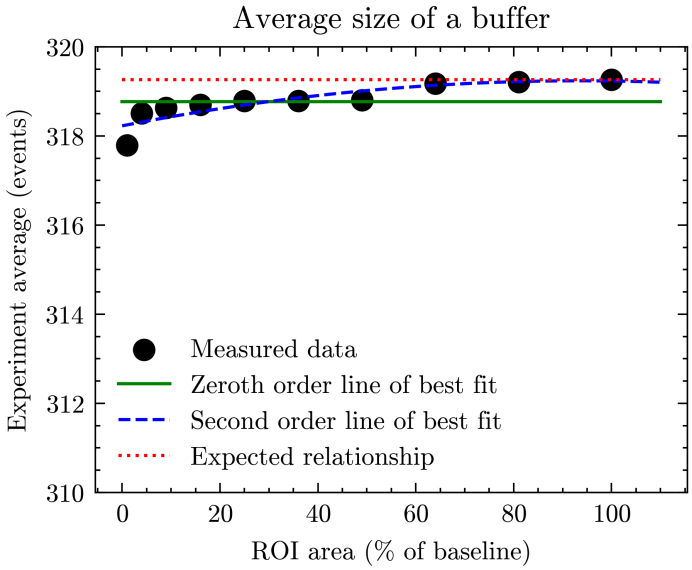

Figure 6.1: Buffer size across all ROI experiments

$$|b| = len(b) \tag{6.7}$$

$$\overline{|b|} = \frac{\sum_0^n |b|}{n} \tag{6.8}$$

Where $|b|$ is the size of the buffer, b is a buffer which is a one dimensional list of events, $\overline{|b|}$ is the mean of the buffer sizes and n is the number of buffers in the experiment.

The zeroth order line of best fit is seen to be slightly lower than the expected constant behaviour due to the slight decrease in measured data as the ROI decreased.
The ratios given in Table 6.2 show us that the buffer size stayed very constant across experiments, and is largely independent of the ROI. The 10% ROI experiment had the largest difference from the baseline of the experiments with a 0.8% decrease, whilst all the other experiments are within 0.5% of the baseline average buffer size.
If the buffer size did not change at all, it would be expected that buffer size stays constant across all experiments. The fact that the buffer sizes do differ, means that the event buffer is adaptive to some degree. The fact that the change is so small, in the order of 2 events, means that the buffer size was very well engineered, not needing large changes to keep performance.
Adaptive in the context of the event buffer means that the number of events that need to be in the buffer before being transmitted out from the RDK to the user changes. In the case where all buffers are found to be the same size, there would be a set limit of events required that was immutable. By looking at the values given in Table 6.1 the average baseline buffer size is roughly 318 events, while the experiment with the lowest average buffer size, 10% ROI, contains 316 events. This difference of only 2 events

makes up less than 1% of the baseline buffer size, and when considering how an ROI of 10% has an area 1% the size of the full 100% ROI, this difference is almost negligible.

6.1.1.2 Time between events

The time between events, or simply ΔT_{event}, is the time difference in microseconds between consecutive events calculated using Equation 6.9. The inconsistency of this time is a type of jitter, a term primarily used in networks. The ΔT_{event} values given in Table 6.1 is the average value across all events within each experiment calculated using Equation 6.10, and represents the overall event jitter of the experiment. Equation 6.10 was used to first calculate the average ΔT_{event} across a single experiment, and then using the experiment averages, was used to calculate the experiment ΔT_{event}.

$$\Delta T_{event} = t_n - t_{n-1} \tag{6.9}$$

$$\overline{\Delta T}_{event} = \frac{\sum_{i=1}^{m} \Delta T_{event}}{m} \tag{6.10}$$

Where t_n is the nth event timestamp in the experiment.
m is either the number of measurements in the run or the experiment, depending on whether $\overline{\Delta T}_{event}$ is being calculated for the run or the experiment.

Table 6.2 show that across the experiments, ΔT_{event} changes drastically, ranging from 97% on the low end with an ROI of 100%, up to a ΔT_{event} 917.1% that of the baseline experiment for the 10% ROI experiment. The difference between an ROI of 10% and 100% of nearly 10 times, while the difference between 20% and 100% is only 4 times.

What is important is that these large deviations still match with the expected behaviour. ΔT_{event} is expected to be related to the pixel height of the experiment as explained in Section 6.1. Shown in red is the expected behaviour which was created using the function described in Equation 6.3.
The fit of this curve can be seen to be close, supporting the hypothesis that ΔT_{event} is related to the ROI pixel height. This fit was calculated to have a Pearson correlation coefficient (r) of 0.998 which shows the strong correlation. The RMSE for the expected behaviour was found to be 0.068 which is very low for the scale of the data, again showing the accuracy of the model. Comparing the RMSE and Pearson correlation coefficients of the expected behaviour and the line of best fit further supports the expected behaviour of the data.

Figure 6.2 shows the measured ΔT_{event} values, along with the expected behaviour and the line of best fit.

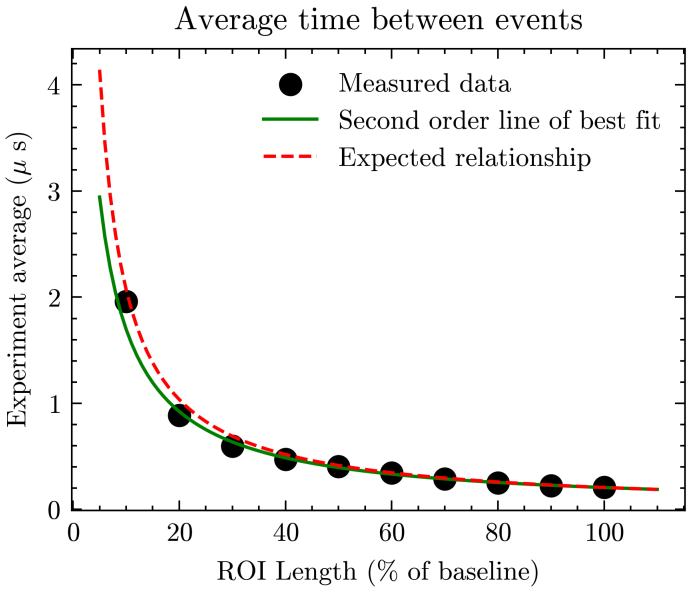

Figure 6.2: ΔT_{event} across all ROI experiments

Figure 6.3 is a histogram of the average experiment ΔT_{event} for the baseline experiment. It has a roughly normal distribution, which is expected. As the ΔT range is very small, from 0.208 to 0.22, the standard deviation is also very small, with the mean at 0.2. The small range and normal distribution shows us that the experiment has little jitter, and that ΔT_{event} when not using an ROI filter can be expected to be constant. This is the ideal case as it shows that the FPGA is passing events to the USB buffer at regular intervals and with inconsequential delay, for the buffers to be sent to the user.

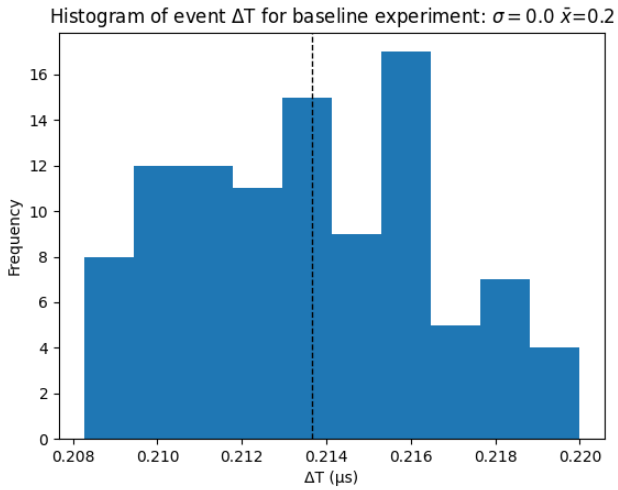

Figure 6.3: Baseline ΔT_{event} histogram

This is further supported by Figure 6.4 which is a spread of 75 events across a typical buffer in the baseline experiment. Although this graph contains 75 events,

only 13 are visible due to fast event rate causing multiple events to share the same timestamps, which I will discuss more in Section 6.1.1.5. The y-axis of this figure only ranges from 16 to 28, with groups of events seen regularly spaced in one timestamp intervals.

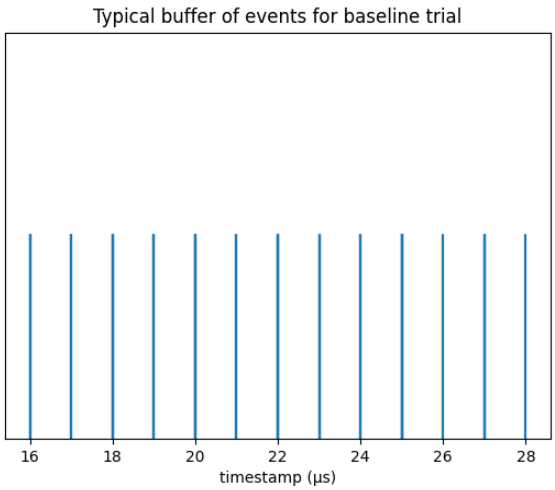

Figure 6.4: Typical buffer of events in baseline experiment

To compare the effect different ROI resolutions have to $x_{\Delta T_{event}}$, the results from the 10% ROI experiment are used as they deviated from the baseline measurements the most. Therefore, the histogram of $x_{\Delta T_{event}}$ for the 10% ROI experiment is shown in Figure 6.5. This figure shows that the distribution is no longer normal, meaning that there can be much less confidence in the expected jitter when using a 10% ROI filter. This is shown with the much higher standard deviation of 3.52 and the much larger range of values, spreading from just over zero to just under 14 μs. With the mean ΔT_{event} being two μs, ignoring jitter, the performance of the system in terms of ΔT_{event} has decreased by a factor of ten. While the time difference is still in the order of two μs, if the camera was being used in high speed, situations which was already pushing the time resolution of camera, this difference would become much more noticeable.
Subplot *a* in Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8 and D.9 in Appendix D show the ΔT_{event} histograms of the other experiments, and it can be seen that as the ROI approaches the baseline resolution, the distribution becomes more normal. Each figure shows the standard deviations and means of the distributions.

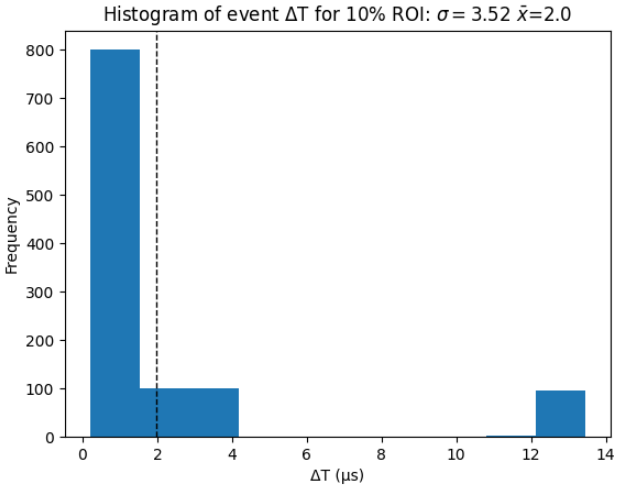

Figure 6.5: 10% ΔT_{event} histogram

Further exploring the jitter of the 10% ROI experiment with Figure 6.6, a typical spread of events appears quite different from a typical baseline buffer. Firstly, more events are visible, meaning that the event rate has decreased as fewer events are sharing timestamps. The individual $x_{\Delta T_{event}}$'s is inconsistent, with a mix of small and quite large gaps. As the jitter is quite large in this experiment, in comparison to the baseline experiment, the FPGA is clearly being delayed by something, causing events to be inconsistently passed to the USB buffer.

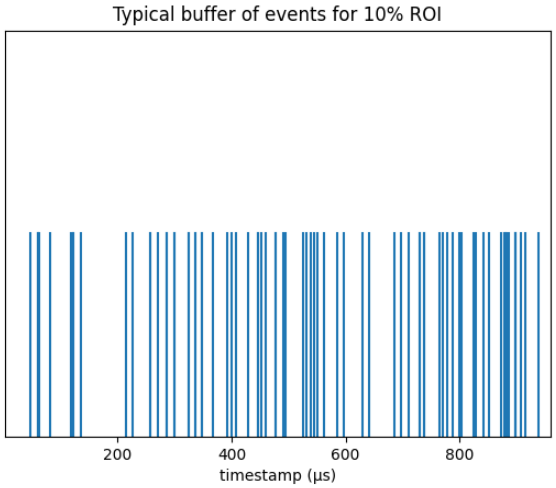

Figure 6.6: Typical buffer of events in 10% ROI experiment

Histograms of the event time differences in the other experiments are shown in subplot e in Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8 and D.9 in Appendix D. These histograms show how the event time differences are effected by the ROI. As the ROI increases towards the full resolution, the histogram reflects Figure 6.3, and as

the ROI decreases towards the 10% ROI, the histograms appear similar in scale and distribution to Figure 6.5.

6.1.1.3 Time between buffers of events (Buffer delay)

Buffer delay in Table 6.1 represents the average time between consecutive buffers across the experiment. As the event camera used has a microsecond time resolution, buffer delay is measured in μs. Along with ΔT_{event} discussed in Section 6.1.1.2, buffer delay is another type of jitter of the RDK. The time between buffers was calculated similarly to the time between events as it was found as the difference between consecutive event timestamps. The events used needed to be the last event in a buffer and the first event in the next buffer, as they are consecutive events that represent the start and end timestamps of consecutive buffers. Equation 6.11 shows the equation used to calculate buffer delay for a pair of buffers. Equation 6.12 shows the equation used to calculate the average across the experiment. The experiment results for buffer delay are shown in Table 6.1.

$$Delay = t_{0_b} - t_{n-1_{b-1}} \tag{6.11}$$

$$\overline{Delay} = \frac{\sum_{i=1}^{m} Delay}{m} \tag{6.12}$$

Where t_{0_b} is the first event timestamp in buffer b and $t_{n-1_{b-1}}$ is the last event timestamp in buffer $b-1$, the buffer that precedes b. m is either the number of buffer delay measurements in the run or the experiment, depending on whether $\overline{Delay}$ is being calculated for the run or the experiment.

Buffer delay deviates very similarly to ΔT_{event} discussed in Section 6.1.1.2. Table 6.2 shows us that with the 100% ROI experiment, the buffer delay is 97.3% that of the baseline buffer delay. The 10% ROI experiment is more than 997.7%, the largest deviation of any measurement in any experiment. As buffer delay was calculated the same as ΔT_{event}, as the difference between consecutive events timestamps, they had very similar values ranging roughly from 0 to 2 as shown in Table 6.1. The true shape of the relationship between ROI and buffer delay is shown when all ROI experiments are plotted, as they are in Figure 6.7.

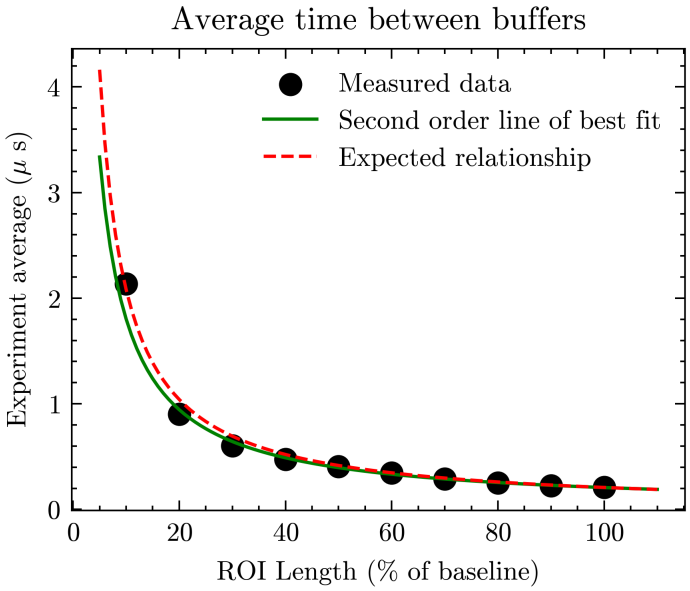

Figure 6.7: Buffer delay across all ROI experiments

Buffer delay is expected to be related to the ROI pixel row height as explained in Section 6.1, similar to ΔT_{event}. Shown in red is the expected behaviour which was created using the function described in Equation 6.3.
The fit of this curve can be seen to be close, supporting the hypothesis that buffer delay is related to the ROI height. This fit was calculated to have a Pearson correlation coefficient (r) of 0.996 which shows the strong correlation. The RMSE for the expected behaviour was found to be 0.057 which is very low for the scale of the data, again showing the accuracy of the model. Comparing the RMSE and Pearson correlation coefficients of the expected behaviour and the line of best fit further supports the expected behaviour of the data.

To understand what could be causing this behaviour, Figure 6.8 provides a view of five typical consecutive buffers and their timestamp ranges in the baseline experiment. These buffers have minimal gaps between them, having at most one μs difference.

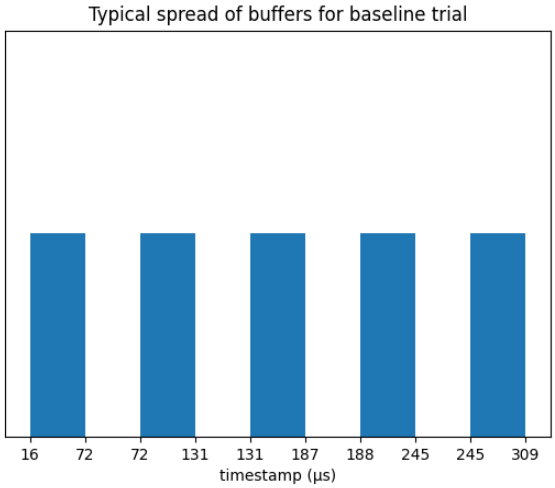

Figure 6.8: Typical buffers in baseline experiment

Figure 6.9 is a typical collection of five consecutive buffers taken from the 10% ROI experiment, as it provides the greatest deviation from the baseline of all the experiments. The time difference between buffers shown here averages nine μs which, while on the lower end, matches the order of magnitude the buffer delay for a 10% ROI has in Table 6.1. However, This average is skewed by an outlier, namely the difference between the first and second buffers. Using a 25% trimmed mean, the mean of this data is actually $\bar{x}_{0.25} = 3.5$, but due to the small experiment size this is an unreliable value.

Typical spreads of buffers from the other nine experiments are shown in subplot f in Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8 and D.9 in Appendix D. The averages from each of these also loosely fit the corresponding buffer delay values in Table 6.1 to a degree that can be expected of by such a small experiment size.

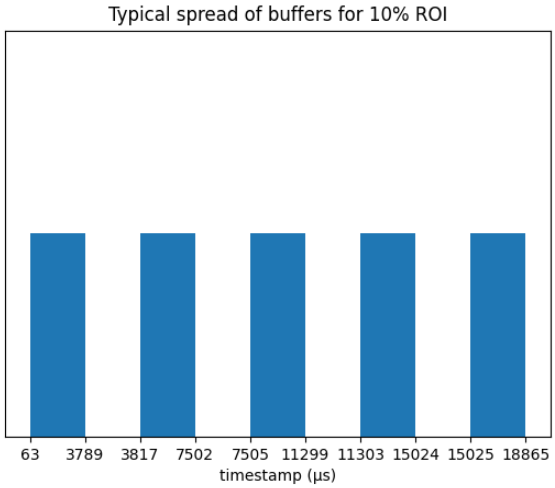

Figure 6.9: Typical buffers in 10% ROI experiment

Figure 6.10 shows the distribution of buffer delays over the baseline experiment. This is clearly a normal distribution around the mean of 0.2 μs with a standard deviation of 0.02. The bins of the histogram range from 0.18 to 0.24 μs, a very small spread of averages which shows the consistency and speed events were transmitted from the RDK during this experiment.
The histogram generated from the 10% ROI experiment, shown in Figure 6.11, however cannot be described as normal. The standard deviation is over 200 times that of the baseline, showing the large unpredictability of the buffer delay. The larger spread of values and distribution explain how while a majority of buffers have little time between each other, a significant number of outliers to this brings the average up, in the order of a factor of two.

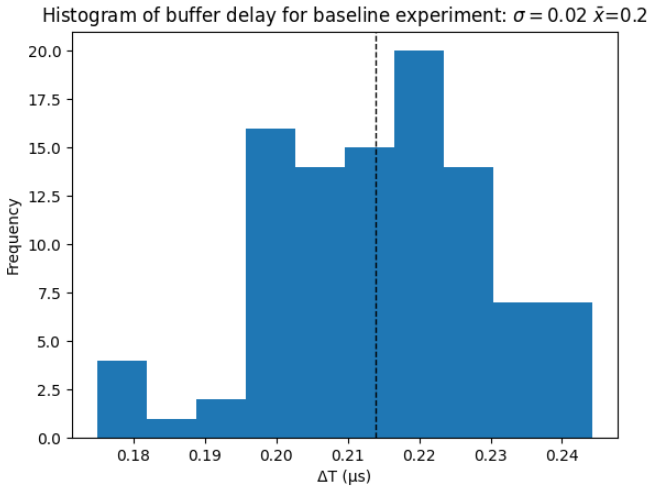

Figure 6.10: Baseline buffer delay histogram

Despite these decreases in performance, the most important finding from this measurement is that with the decrease in ROI, no events are lost between buffer transmissions. It was a possibility that as the ROI decreased and time between events increased, that events could be dropped as the time resolution went up. However, as best shown in Figure 6.9 and comparing this to Table 6.1, the time between buffers matches the expected ΔT_{event}, accounting for outliers. This means that there is nothing different between the time between consecutive events within a buffer, and the last event of one buffer and the first event of the next buffer.

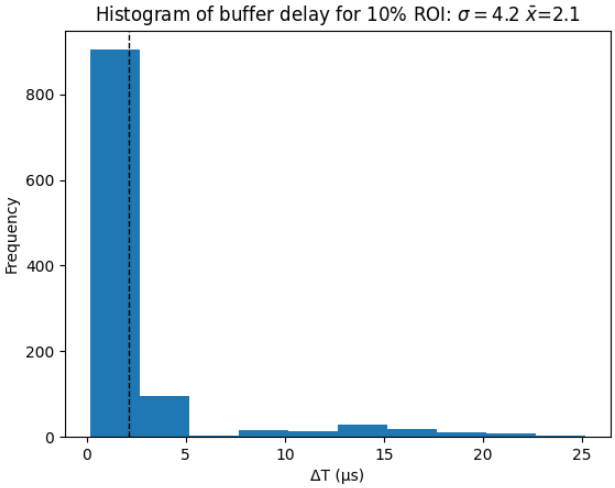

Figure 6.11: 10% buffer delay histogram

Histograms of the buffer delays in the other experiments are shown in subplot *b* in Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8 and D.9 in Appendix D. These histograms show how the buffer time differences are effected by the ROI. As the ROI increases towards the full resolution, the histograms reflect Figure 6.10 and as the ROI decreases towards the 10% ROI the histograms appear similar in scale and distribution to Figure 6.11.

From the perspective of the FPGA, ROI does not affect the incoming event stream. The jitter as discussed here and in Section 6.1.1.2, is most likely caused by the dumping of events. Despite the FPGA deleting events, timestamps are still assigned using the same row based algorithm described in Section 3.1.

6.1.1.4 Buffer time span

The time span across a buffer, or (ΔT_{buffer}), is the difference in timestamps between the first and the last events in the buffer. This is the earliest and latest events, or more simply the accumulation time of the buffer. This is an important measurement as it shows the delay in the system at the level of the buffer, which is more noticeable than an event by event basis. Equation's 6.13 and 6.14 were used to calculate the experiment average and the experiment average respectively.

$$\Delta T_{buffer} = t_{n-1_b} - t_{0_b} \tag{6.13}$$

$$\overline{\Delta T}_{buffer} = \frac{\sum_{i=1}^{m} \Delta T_{buffer}}{m} \tag{6.14}$$

Where t_{0_b} is the first event timestamp in buffer b and t_{n-1_b} is the last event timestamp in buffer b.

m is either the number of measurements in the run or the experiment, depending on whether $\overline{\Delta T}_{buffer}$ is being calculated for the run or the experiment.

ΔT_{buffer} is the last measurement in Table 6.1 that has a negative relationship with ROI. In Table 6.2, ΔT_{buffer} has very similar results to ΔT_{event}, even compared to buffer delay. This shows how ΔT_{buffer} is closely related to ΔT_{event}. As buffers average 318 events as seen in Section 6.1.1.1, the experiment values for ΔT_{buffer} are seen as roughly the product of ΔT_{event} and buffer size. This makes theoretical sense as ΔT_{buffer} is equivalent to the time between two events exactly one buffer size apart.
The experiment ΔT_{buffer} value was calculated as the average of all ΔT_{buffer} across the runs. Rounding and non-normal distributions of ΔT_{buffer} is what causes the difference between the measured ΔT_{buffer} and the theoretical ΔT_{buffer} calculated using ΔT_{event} and buffer size.
Figures 6.12 and 6.13 show the distributions of ΔT_{buffer} for the baseline experiment and the 10% ROI experiment respectively. The other experiment ΔT_{buffer} distributions are shown in subplot c in Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8 and D.9 in Appendix D.

The distribution of ΔT_{buffer} for the baseline experiment is relatively normal, with an average of 68 μs and a standard deviation of $\sigma_{baseline} = 0.98\mu s$. However, the ΔT_{buffer} distribution of the 10% ROI experiment has a mean of 611.8 μs and a standard deviation of $\sigma_{10\%} = 1086.42$, an increase of more than 1100%. This difference shows that the received event stream in terms of the ΔT_{buffer}, is effected greatly by the decrease in ROI negatively. ΔT_{buffer} is too unpredictable in timing, sometimes taking much longer to pass an event buffer to the user. This would be further affected by the distribution of events across the cameras focus. If there were little events occurring for the camera to capture, it is likely that ΔT_{buffer} would increase even more, and potentially decrease in situations with many events.
The distributions across all experiments becomes more normal, and hence more predictable, as the ROI approaches the baseline resolution.

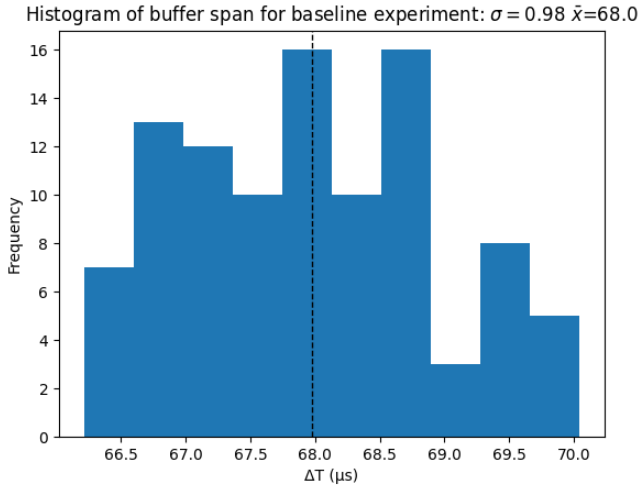

Figure 6.12: Baseline ΔT_{buffer} histogram

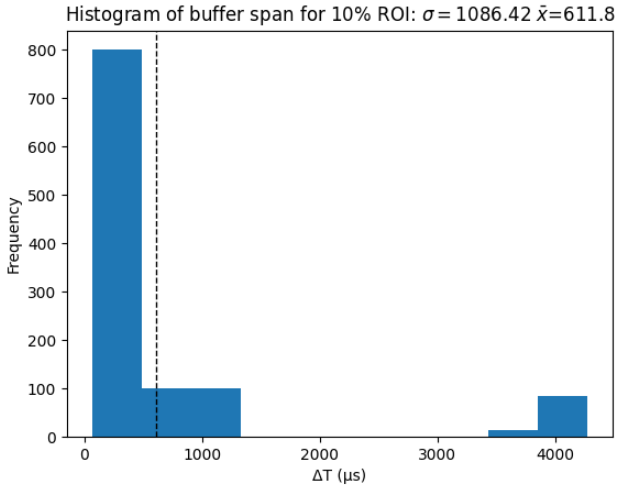

Figure 6.13: 10% ΔT_{buffer} histogram

ΔT_{buffer} is expected to be related to the ROI pixel height as explained in Section 6.1 similar to ΔT_{event}. Shown in red is the expected behaviour which was created using the function described in Equation 6.3.

The fit of this curve can be seen to be close, supporting the hypothesis that ΔT_{buffer} is related to the ROI height. This fit was calculated to have a Pearson correlation coefficient (r) of 0.998 which shows the strong correlation. The RMSE for the expected behaviour was found to be 24.004 which is very low for the scale of the data, again showing the accuracy of the model. Comparing the RMSE and Pearson correlation coefficients of the expected behaviour and the line of best fit further supports the expected behaviour of the data.

Figure 6.14 shows the measured ΔT_{buffer} values, along with the expected behaviour and the line of best fit.

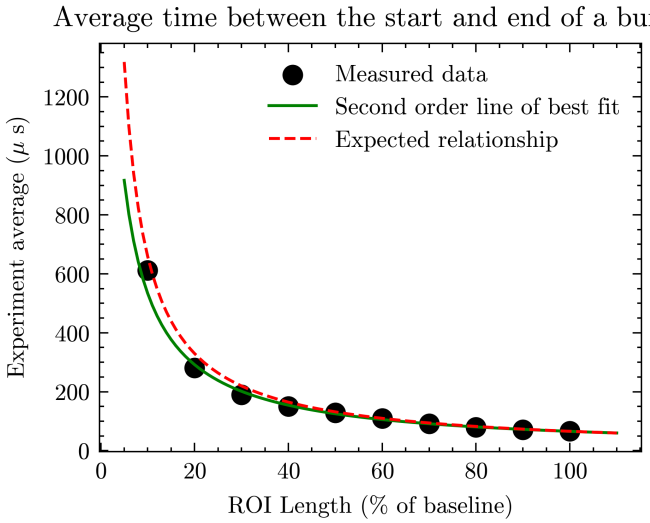

Figure 6.14: ΔT_{buffer} across all ROI experiments

The relationship ΔT_{buffer}, ΔT_{event} and buffer delay have with the ROI pixel height is due to the way the event camera detects and assigns events. As described in Section 6.1.1.3, this comes from the way the cameras row arbiter indexes through pixels and assigns timestamps. The cameras' arbiter works the same, despite the filter, as the ROI filter is integrated into the FPGA. Events are simply ignored and not passed to the client if outside the ROI. However, these pixels still produce events in the camera and are passed to the FPGA. The arbiter still spends a finite amount of time indexing through all pixels, regardless of whether they are ignored in the FPGA or not.

6.1.1.5 Event rate

The event rate here is the rate at which events fill a buffer and is measured in events per μs. This value is calculated over one buffer, using the buffer span, as discussed in Section 6.1.1.4, and the number of events in the buffer, discussed in Section 6.1.1.1. Individual buffer event rates were calculated using Equation 6.15, and Equation 6.16 was used to calculate the average run and experiment event rates.

$$Rate_{event} = \frac{size_b}{\Delta T_{buffer}} \tag{6.15}$$

$$\overline{Rate}_{event} = \frac{\sum_{i=1}^{m} Rate_{event}}{m} \tag{6.16}$$

Where $size_b$ is the size of the buffer defined in Equation 6.7. ΔT_{buffer} is the difference in time of the first and the last event in the buffer, defined in Equation 6.11.
m is either the number of measurements in the run or the experiment, depending on

whether $\overline{\Delta T}_{buffer}$ is being calculated for the run or the experiment.

Event rate is seen to decrease as the ROI of the camera also decreases in Table 6.1, unlike ΔT_{event}, ΔT_{buffer} and buffer delay which all increased as ROI decreased. Figure 6.15 shows the experiment event rates, including the expected behaviour, and the first and second order lines of best fit.

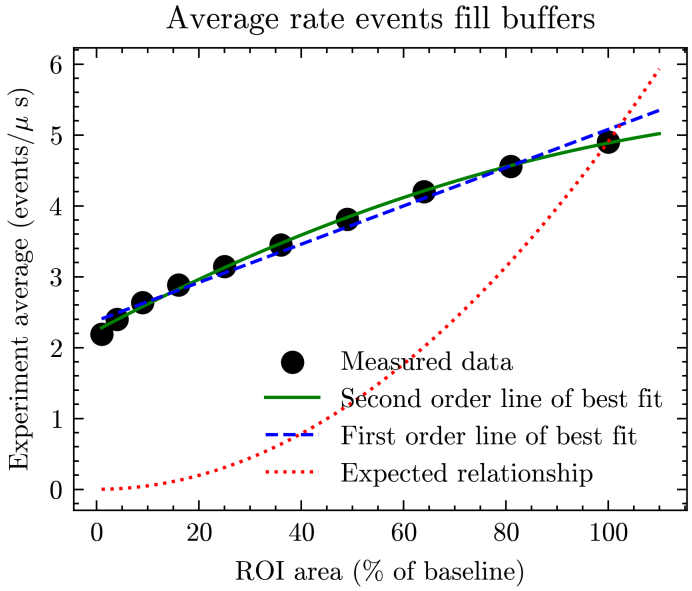

Figure 6.15: Event rate across all ROI experiments

The event rate is expected to be related to the pixel area of the experiment as explained in Section 6.1 and given by the function in Equation 6.2. Shown in red is the expected behaviour which was created using the function described in Equation 6.2. This behaviour deviates from the measured data, which means that there is an effect other than the ROI area that effects the event generation. To deviate as is seen, the event camera must be generating more pixels as the ROI area decreases. The scenes between experiments were kept similar. This means that it does not come from different experiments experiencing different amounts of light. It is possible that the culling of events occurs within either the event camera, or within the FPGA. To further study this effect, data output from the event camera needs to be compared to the data output from the FPGA. This will expose where this effect originates.

Looking at the measurements in Table 6.2, a 20% ROI, which has an area 4% that of a 100% ROI, decreases the event rate by 50%. The event rate decreases with ROI and does so at a rate of approximately 0.3 events per μs per 10% ROI change. As event rate is calculated using the size of a buffer and the number of events in the buffer, as shown in Equations 6.15 and 6.16, they should be reflected in Table 6.1 using buffer size

and ΔT_{buffer}. As these values are experiment averages there is some difference between the values that can be calculated here and what is reported in the table. At higher ROIs, the reported event rate matches closely the approximate event rate calculated using the other measurement averages. As the ROI decreases however, this accuracy decreases down to the point where for the 10% experiment the values differ by a factor of 4. This reinforces the wide range of ΔT_{buffer} values shown in subplot c in Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8 and D.9 in Appendix D.

Looking at the distribution of event rates over the baseline experiment in Figure 6.16, the mean rate is $\bar{x} = 4.7\mu s$ and the standard deviation is $\sigma = 0.07$. The distribution is close to normal, ranging from approximately 4.6 to 4.9 μs. Using the 10% ROI experiment, Figure 6.17 shows the distribution of event rates over the experiment. The shape of the distribution is far from normal, instead showing a multi-modal distribution. This means that there is likely to be more error if it was used as a predictive model for future event rates, especially over longer periods of time. The distributions of the event rates of the other experiments is shown in subplot d in Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8 and D.9 in Appendix D.
The mean rate for the 10% ROI experiment of $\bar{x} = 2.2$, is less than half that of the baseline. The standard deviation also increased by a factor of 24, up to $\sigma = 1.69$. A larger σ means that the event rates across the experiments vary more than in the baseline experiment.

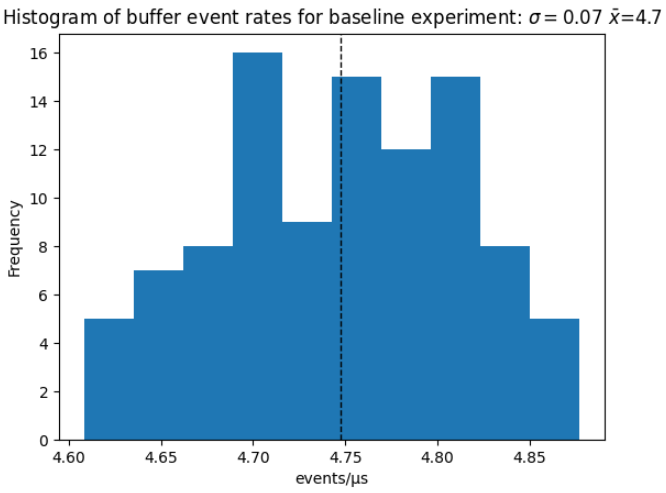

Figure 6.16: Baseline event rates histogram

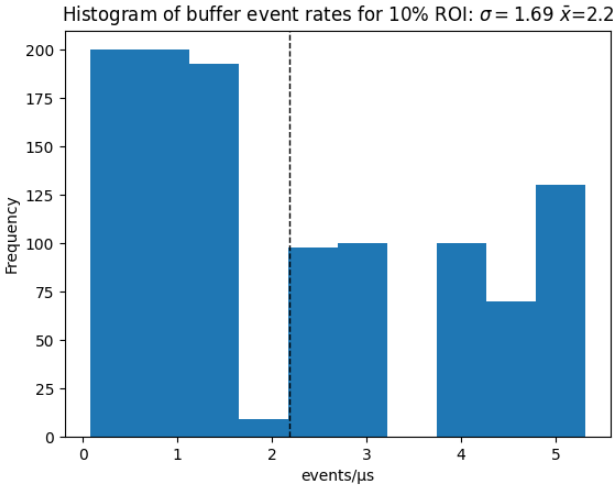

Figure 6.17: 10% event rates histogram

6.1.1.6 Total number of events in the experiment

The total number of events collected is expected to be similarly related to the ROI area as the event rate, explained in Section 6.1 and given by the function in Equation 6.2. As the area decreases the number of events collected is expected to decrease at the same rate. The first order line of best fit in Figure 6.18 confirms this. Shown in red is the expected behaviour which was created using the function described in Equation 6.2. This behaviour deviates from the measured data, which means that there is an effect other than the ROI area that effects the event generation. The possible cause of this effect is discussed further in Section 6.1.1.5. To understand the cause of the effect, the same method as described in Section 6.1.1.5 is needed.

Tables 6.1 and 6.2 give the precise measurements calculated, which are represented in Figure 6.18.

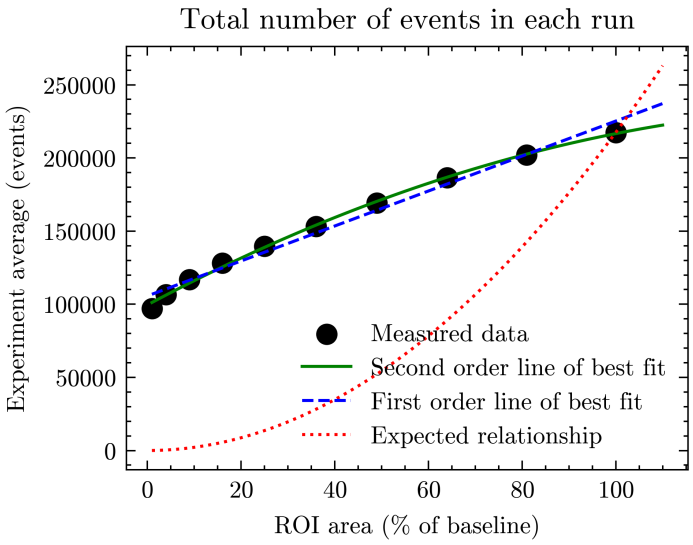

Figure 6.18: Total number of events across all ROI experiments

Conclusion

A native host installation of the software applications *Vivado* and *PetaLinux* is the best solution for this projects work environment, as discussed in Chapter 4. The complexity of a virtualisation solution is unrealistic and too complicated when considering the purpose of the project.

The results in Chapter 6 demonstrate that the three latency measurements behave as described in Section 6.1, specifically Equations 6.1 and 6.3. The three latency measurements are: the time between events, the time between buffers and buffer delay from the perspective of the FPGA. This was expected as the event camera iterates through the pixel array, regardless of the ROI, as the ROI is implemented at the FPGA level. As the FPGA discards events from pixels outside the ROI, it is expected that gaps in timestamps appear. This is due to the camera taking a finite amount of time to assign timestamps to pixels, as described in Section 6.1.
The event rate and total number of events captured during the experiments were expected to have a relationship inversely related to the latency measurements, which is given in Equation 6.2, but were found to deviate. This is most likely due to either the event camera or the FPGA deleting events based on the amount of events being registered. The expectation was that as the latency measurements are inversely related to the ROI pixel row height, and for event rate and the number of events to be proportional to the ROI area. The total number of events is easy to understand as a decrease in pixels that events can come from would decrease the number of events at the same rate.
Lastly, the buffer sizes were found to decrease slightly with the ROI area, but stay largely constant. This clearly shows that the default buffer is adaptive, influenced by the ROI. The scale at which the buffer sizes change however is very small, demonstrating that the size was finely tuned, further discussed in Section 6.1.1.1.
The experiments explained in Chapter 5 each contained 100 individual runs, minimising sampling error. While the measurements found for each experiment had low error, as there were only 10 ROI experiments conducted, this project was limited to this data. Discussed in Section 3.2 and shown with the total number of events in Section 6.1.1.6, reducing the FPGA ROI in the RDK decreases the amount of data that needs to be

processed by the user. The latency measurements show that the performance of the system decreases across an almost constant length of time, as shown by the buffer size discussed in Section 6.1.1.1.
However, from the perspective of the user, the performance metrics from the perspective of the FPGA do not affect the performance metrics of the user. Latency variables such as the time between events values are internal values to the FPGA time axis shown in Figure 3.4. As described in Equation 3.2, the overall latency performance as seen by the user is only affected by the buffer duration (the time between buffers) from the FPGA.
The results presented in Chapter 6 describe the effect of decreasing the camera ROI. It also shows that the output data similarly decreases. The time between buffers is directly affected by the size of the buffer, which was found to stay constant. The overall latency from the user's perspective can be decreased by implementing an ROI filter and configuring a smaller USB buffer size. This will decrease the time between buffers and user's latency, thus decreasing the FPGA latency and the amount of data the user would need to process.

The research presented in this thesis contributes to the event camera research field by describing a more effective way to use event cameras for object tracking purposes. Real-time object tracking will benefit from a decreased amount of data as the processing time will decrease and the response rate of the tracker will increase.

7.1 Future Developments

This section will discuss further areas of study that were not pursued in this thesis. This will include the next step for furthering the study conducted in Section 7.1.1. Section 7.1.2 discusses alternative possibilities to altering the Region of Interest (ROI). Lastly, Section 7.1.3 discusses the importance of the findings of this thesis, and the possible applications for this research and any advances that could come from it.

7.1.1 The Next Step

The next logical step would be to look at the influence of an adaptable event buffer size on performance, as mentioned in Chapter 7. As discussed in Sections 6.1.1 and 6.1.1.1, the event buffer output from the RDK stays constant, despite the ROI. The buffer size had a direct impact on the time span of a buffer from in Section 6.1.1.4, and the event rate from Section 6.1.1.5. These measurements may benefit from the alteration of the buffer size.
The buffer size is a value configured for the USB, and is most likely defined in multiple

places, namely within the USB driver and within a memory register the FPGA can access. It is possible that this value is defined only once, such as in a memory register, and is accessed by both the USB driver and the FPGA. Reducing this value is expected to reduce the latency of the user, leading to improved performance.

7.1.2 An Alternative

It is possible that an ROI based solution may not meet the performance requirements of the user, even with improved performance through an adaptive buffer, as discussed in Section 7.1.1. This thesis suggests 2 possible alternatives to the current USB transfer method, namely a direct wire connection with General-Purpose Input/Output (GPIO) pins, and bit banging.

USB is a general purpose technology, making it quite complex due to its standards and procedures. Due to the time requirements involved in real-time object tracking, this thesis suggests that the best alternative to the USB connection is a direct pin connection between the RDK and the user. This bypasses the USB driver stack, greatly reducing transfer complexity and the transfer delay from the FPGA, given as $\Delta\tilde{t}$ in Equation 3.2.

Bit banging is primarily done using GPIO's, but it is also possible to bit bang the USB protocol using the existing hardware [44]. The main advantage gained through bit banging compared to the current USB connection is that a bit banging algorithm could be configured to send events individually. This would reduce the buffer accumulation time, given as ΔT in Equation 3.2, to the processing delay of a single event.

7.1.3 Applications

The decrease in the amount of processing that occurs, both within the FPGA and in any algorithms run by the user, decreases power consumption of the overall system. One benefit to a decreased amount of data that needs to be processed is that any processing algorithms become faster. Decreased data could be used to either speed up existing systems or subsystems, or to integrate multiple systems that were previously not efficient enough to operate together. For example, multiple event camera systems could be used together for fast object tracking from different angles, improving state estimation algorithm accuracy. The benefits extend beyond object tracking, for example, multiple cameras could also be used in real-time three-dimensional modelling or reconstruction of objects.

The main benefits of an ROI filter is evident when there is a single object to be

tracked, and that object has a slow relative motion to the sensor, as discussed in Section 1. The ROI can be set the smallest size in this situation. However, in a multiple object tracking algorithm, an ROI cannot be set this small, even if the targets are slow moving and small. This depends on the position of the targets as a single ROI must include all the targets. If the targets are spread across the full sensor resolution, the ROI must be the same size as the full resolution, leading to unused pixels and data. As most systems do not have one target, this reduces the benefits shown of a single ROI filter. Implementing a system with multiple ROI's would eliminate any unused pixels, so that complex object tracking algorithms would also benefit.

Experimental Setup

Figures A.1, A.2, and A.3 show the stage used in the experiments discussed in Section 5.
Figures A.4 and A.5 show the position of the event camera in relation to the stage for the polarity flip experiment, discussed in Section 4.4, and the ROI experiments, discussed in Section 5, respectively.

Figure A.1: Experimental stage setup

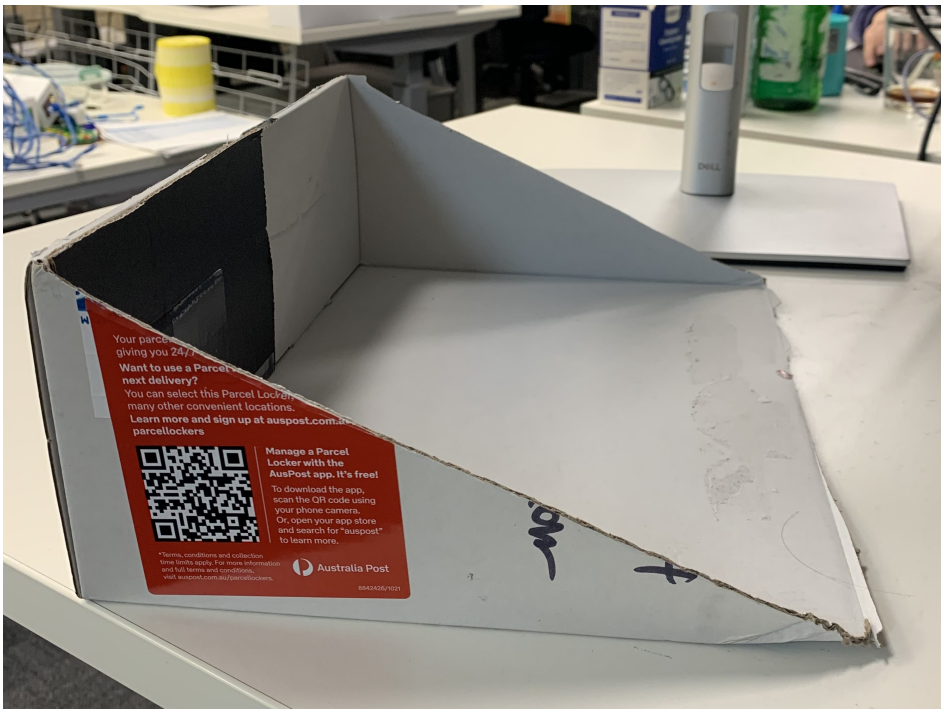

Figure A.2: Side view of experimental stage setup

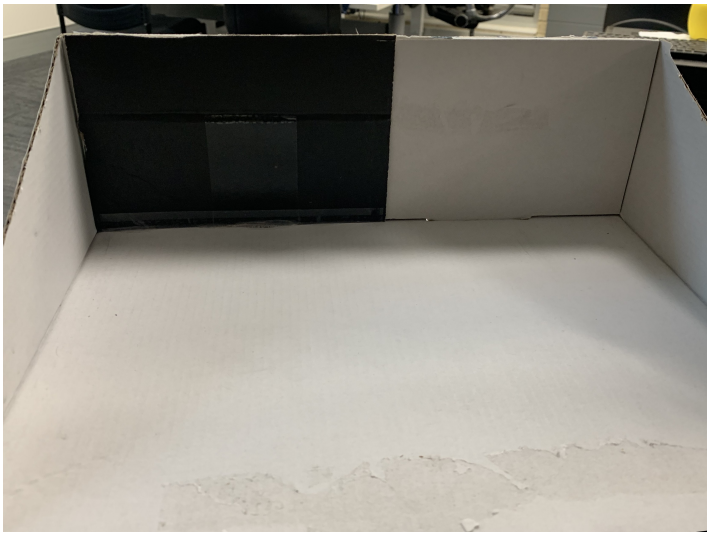

Figure A.3: Black and white target of experimental stage setup

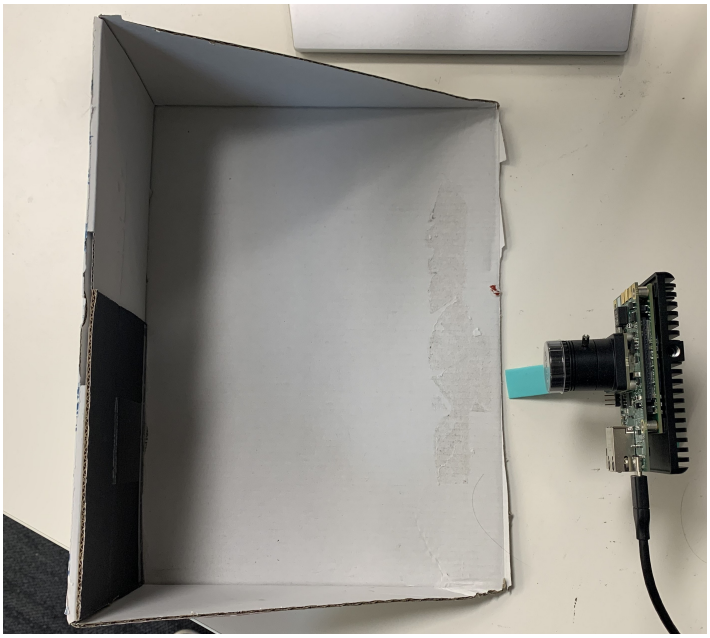

Figure A.4: Experimental setup showing the placement of the RDK within the stage during the polarity experiment

Figure A.5: Experimental setup showing the placement of the RDK within the stage during the ROI experiments

Appendix B

RDK System

Figure B.1 is a detailed block diagram of the entire RDK2 system.
Figure B.2 is a detailed block diagram of the *huahine* block within the *Prophesee* licensed FPGA bitstream.
Figure B.3 is a basic block diagram of the entire system that includes the RDK2 and the device user.

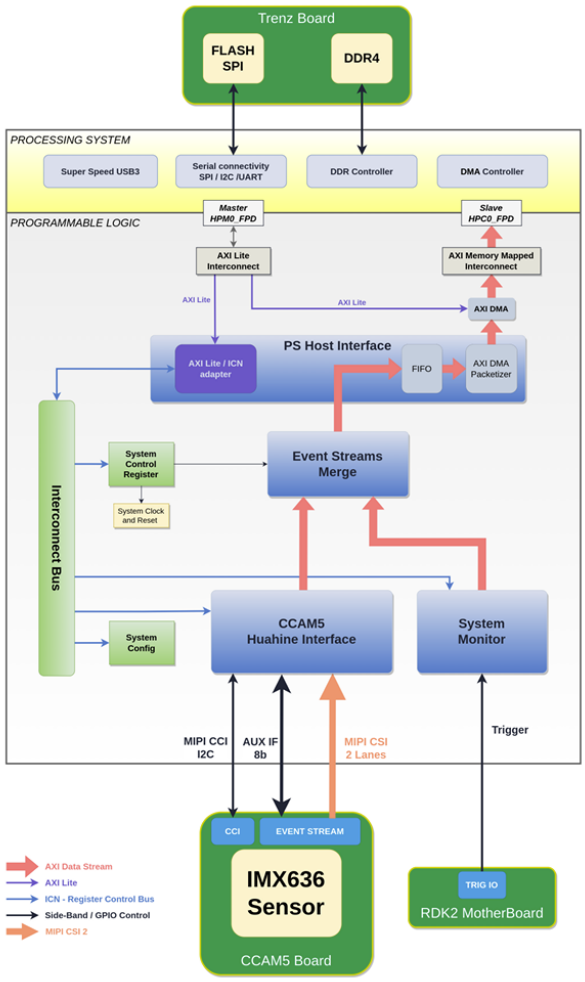

Figure B.1: System block diagram for the *Prophesee* RDK showing the CCAM5 module, the FPGA and its sub-modules and the *Trenz* board representing the RDK system after the FPGA [5]

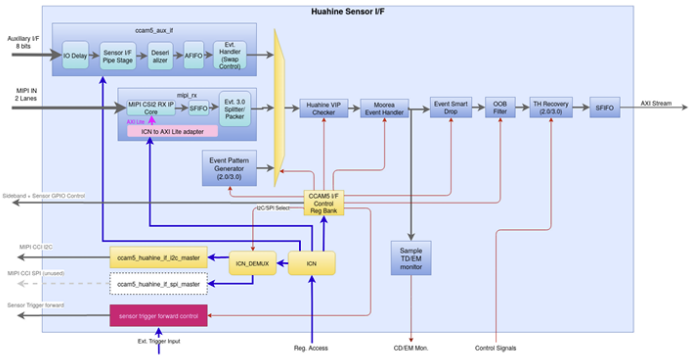

Figure B.2: Block diagram for the main processing block inside the FPGA [5]

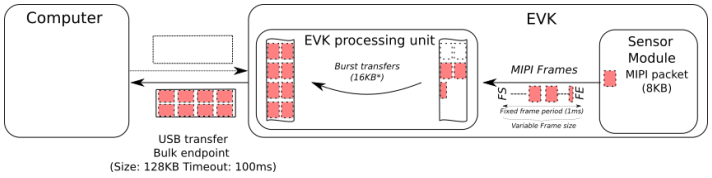

Figure B.3: High level system view of the RDK, broadly showing the path events take from detecting to output [3]

Scripts

The GitHub repository that contains these scripts is found here [45].

C.1 Region of Interest C++ script

This script collects event data from the RDK over a set number of runs for set durations. C++ was chosen to reduce any possible processing delays that could come from a non-compiled language such as Python. This script makes use of *Prophesee* API's to initialise the event camera and access the events.

The general flow of the script is best shown in Figure C.1.
Firstly the script is started, initialising the camera using *Prophesee* API's. Two threads are then started, one that reads the incoming event buffers and creates a csv file containing timestamp information for each event. The second thread acts as a stopwatch, monitoring the run duration. Until the timer exceeds the specified duration of 6.5 seconds, the analysis thread continues. Once the duration is exceeded, a sample counter is incremented by one and the camera is reinitialised, and the process starts again. Once the sample counter reaches 100 samples, the script ends.

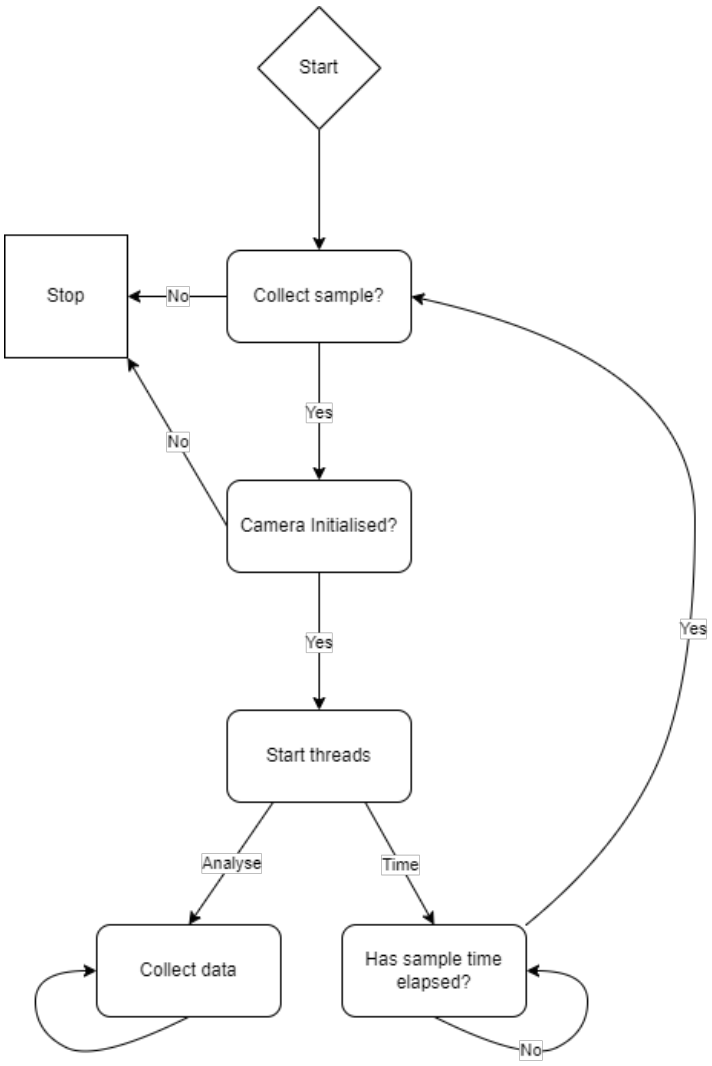

Figure C.1: State diagram for C++ sample collection script

C.2 Region of Interest Python analysis script

This script reads csv files generated by the C++ ROI script (Section C.1), analyses the data and generates tables, plots and histograms of various measurements. This script calculates six measurements from the data, each of which are talked about in detail in Section 6.1.1.

The process is described in Algorithm 1. This is pseudocode describing the flow of the python analysis script.

The script writes a *json* file that contains the calculations for readability and ease of use.

Algorithm 1 Python analysis procedure

```
1: procedure MAIN()
2:     number_of_runs ← 100
3:     dictionary_of_results ← {}
4:     runs ← [Baseline, ROI]
5:     for experiment in runs:
6:         measurements ← calculate_measurements().
7:         dictionary_of_results ← measurements.
8:         generate_plots(measurements);
9:     write_json(dictionary_of_results);
```

C.3 Python table generation and experiment trend plots

This uses the *json* file output from the script in Section C.2 to generate plots to visualise trends over experiments, as well as Latex specific tables for use in this thesis. Tables and plots were generated for each measurement described in List 6.1.1.

The process is described in Algorithm 2. This is pseudocode describing the flow of this script.

Algorithm 2 Python table and plot generator procedure

```
1: procedure MAIN()
2:     results ← read_json_file()
3:     generate_main_latex_table(results);
4:     ratios ← ratios_of_results(results)
5:     generate_latex_table_from_ratios(ratios);
6:     for measurement in results:
7:         generate_plot_from_ratio(measurement_ratio);
```

C.4 Python image magnify procedure

This magnifies a certain area in an image for readability.

The process is described in Algorithm 3. This is pseudocode describing the flow of this script.

Algorithm 3 Python image magnify procedure

```
1: procedure MAIN()
2:     for image in list_of_images:
3:         open_image(image);
4:         magnify_area(image);
5:         save_image(image);
```

Additional Results

Table D.1 shows the Pearson correlation coefficients between the measured data and estimated and second order line of best fit models.

Table D.1: Table of Pearson correlation coefficient values

	Pearson correlation coefficient (r)	
Measurement	**Expected**	**Best fit**
ΔT_{event}	0.998	0.995
ΔT_{buffer}	0.998	0.995
Buffer delay	0.996	0.994

Table D.2 shows the root mean squared error between the measured data and estimated and second order line of best fit models.

Table D.2: Table of RMSE values

	RMSE	
Measurement	**Expected**	**Best fit**
Buffer size	0.645	0.407
ΔT_{event}	0.068	0.085
ΔT_{buffer}	24.004	25.273
Buffer delay	0.057	0.107

Figures D.1, D.2, D.3, D.4, D.5, D.6, D.7, D.8 and D.9 show the different measurements take at 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90% and 100% ROI respectively. Subplots *a*, *b*, *c* and *d* in each figure shows the experiment histograms of the time between consecutive events, time between consecutive buffers, the time span of a single buffer and the event rates respectively. Subplots *e* and *f* show typical spreads of events and buffers within the experiment respectively.

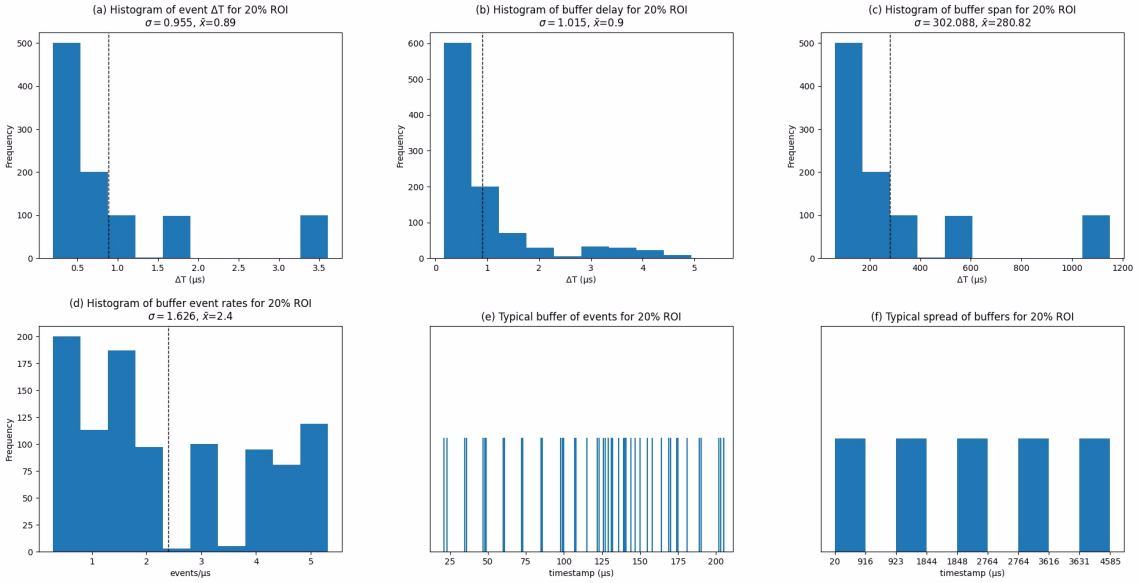

Figure D.1: 20% ROI measurements

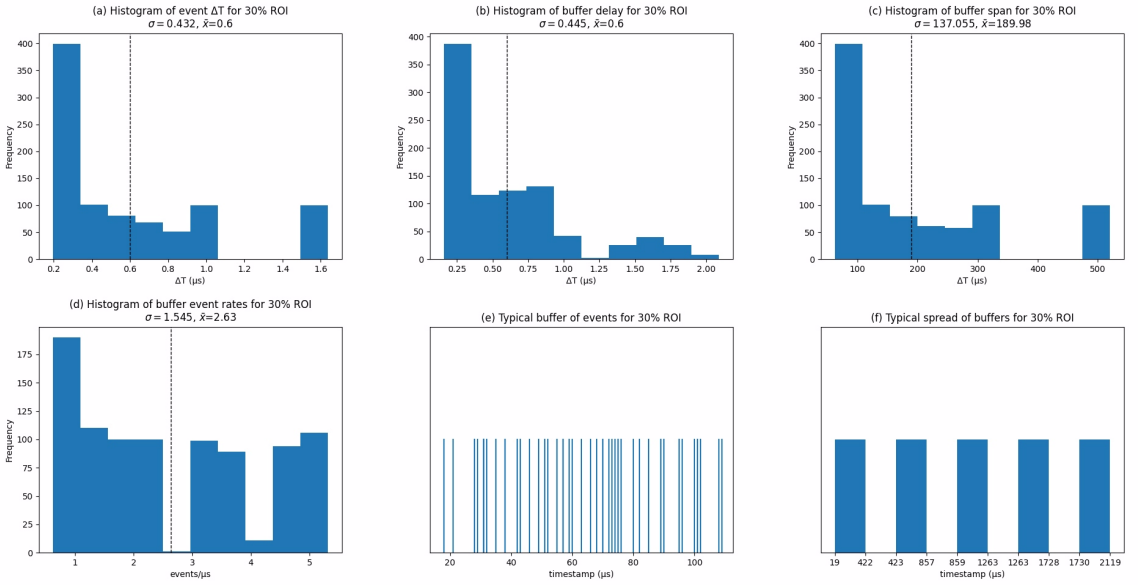

Figure D.2: 30% ROI measurements

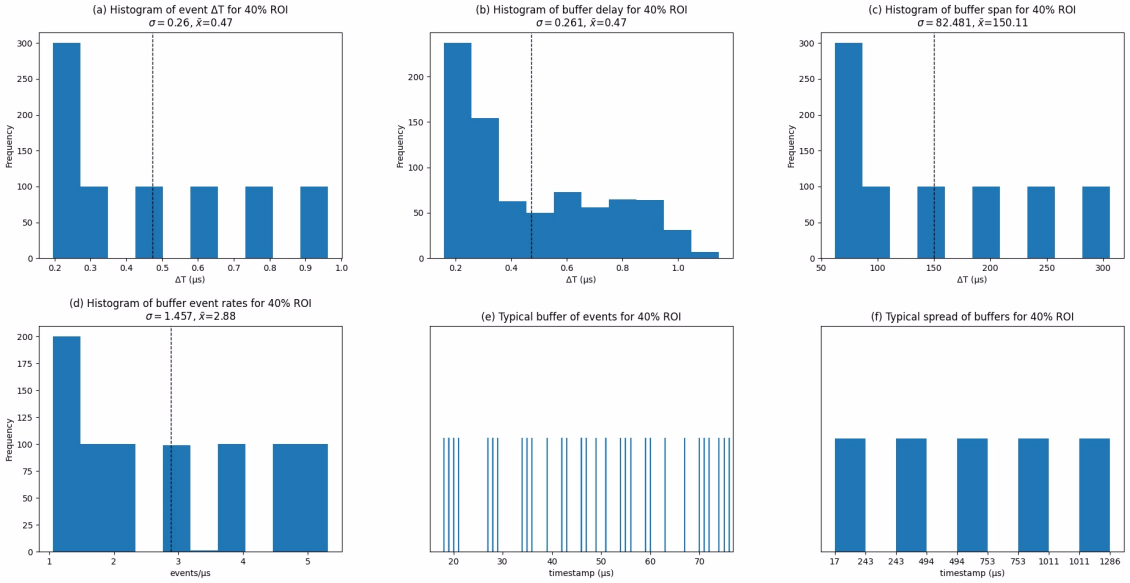

Figure D.3: 40% ROI measurements

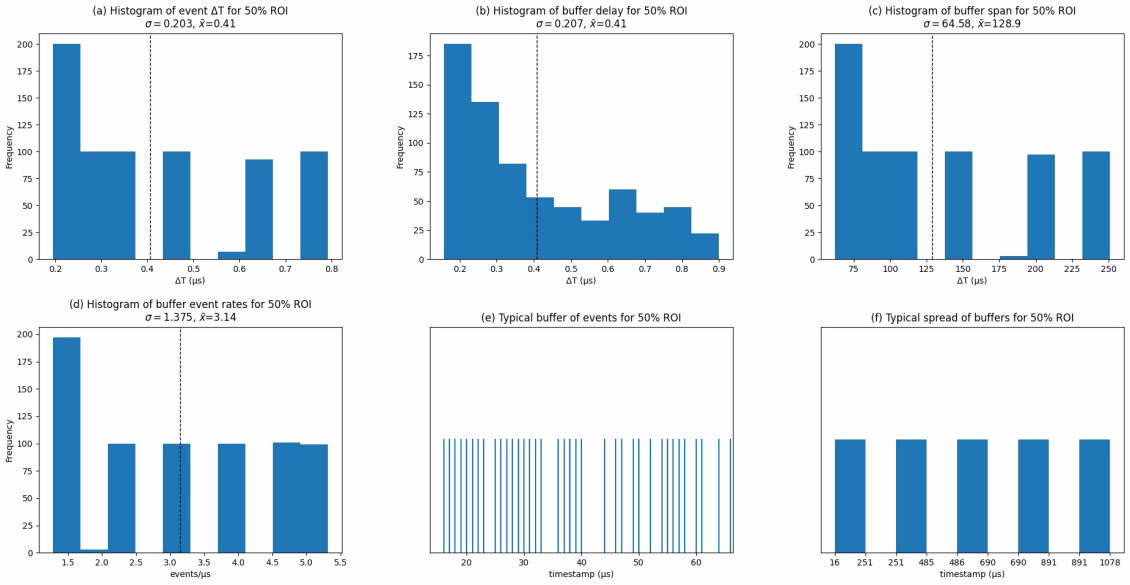

Figure D.4: 50% ROI measurements

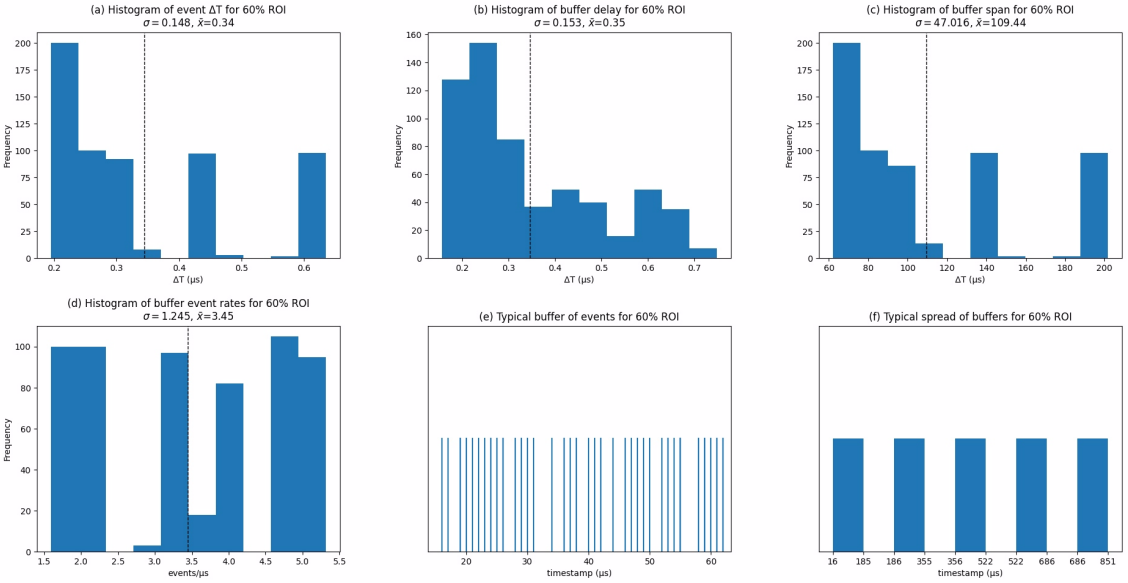

Figure D.5: 60% ROI measurements

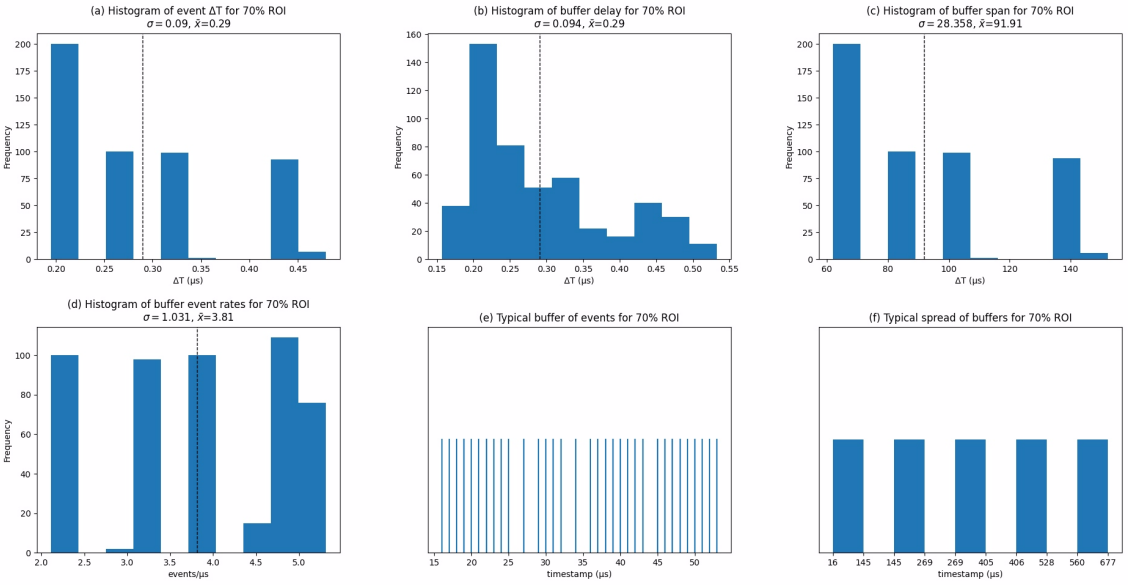

Figure D.6: 70% ROI measurements

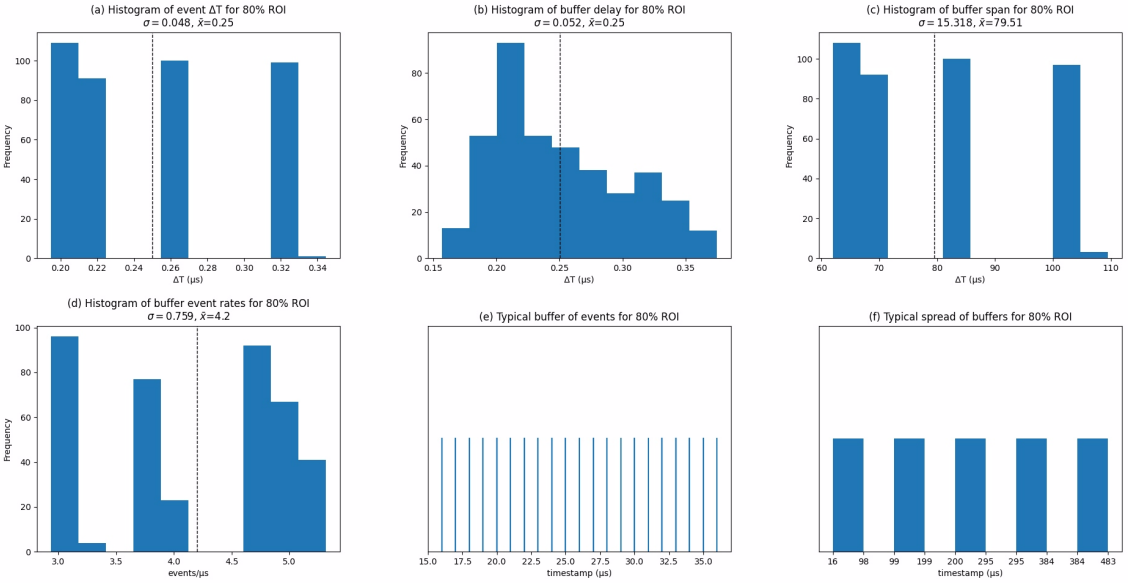

Figure D.7: 80% ROI measurements

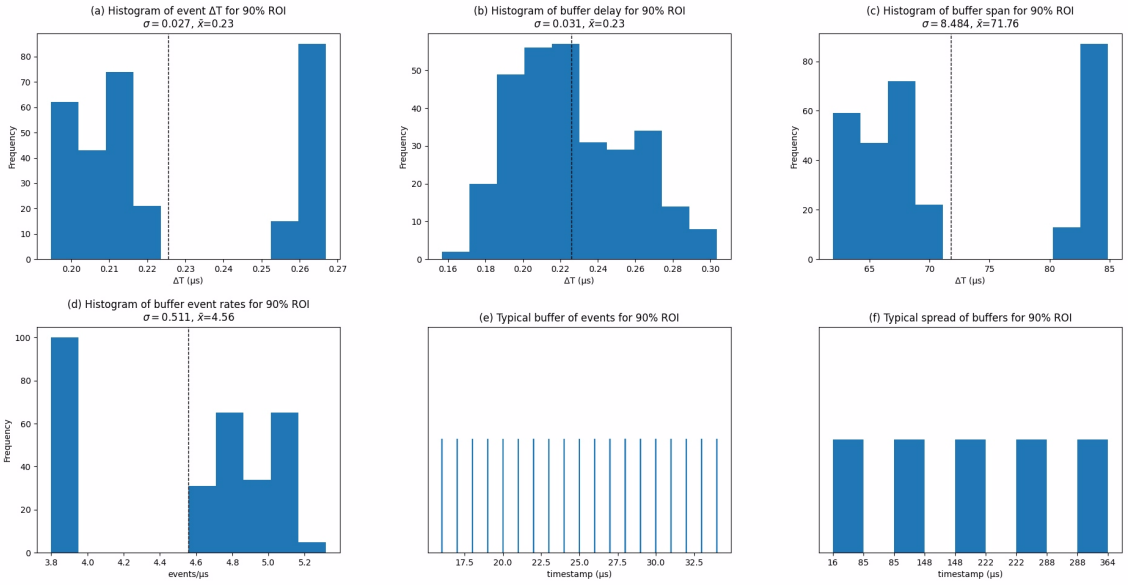

Figure D.8: 90% ROI measurements

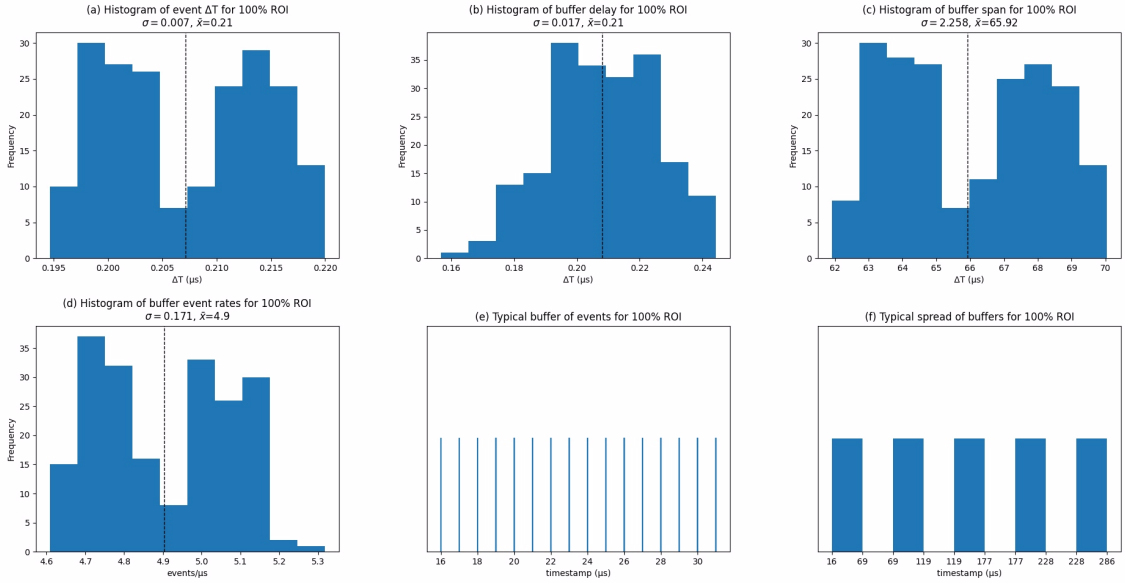

Figure D.9: 100% ROI measurements

Bibliography

[1] Sony-Semicon. Event-based vision sensor (evs). https://www.sony-semicon.com/en/products/is/industry/evs.html, n.d.

[2] Sony-Semicon. Event-based vision sensor (evs). https://www.sony-semicon.com/en/technology/industry/evs.html, n.d.

[3] Prophesee. Transfer latency, 2024.

[4] BaslerWeb. Image roi. https://docs.baslerweb.com/image-roi, 2024.

[5] Prophesee. *RDK2 Technical Reference Manual*, July 2023.

[6] IBM. What are containers? https://www.ibm.com/topics/containers, 2023.

[7] Cisco. What are containers? https://www.cisco.com/c/en/us/solutions/cloud/what-are-containers.html, 2019.

[8] Donald Firesmith. Virtualization via containers. https://insights.sei.cmu.edu/blog/virtualization-via-containers/, 2017.

[9] IBM. What are virtual machines? https://www.ibm.com/topics/virtual-machines, 2023.

[10] Oracle. What is a virtual machine? https://www.oracle.com/au/cloud/compute/virtual-machines/what-is-virtual-machine/, 2023.

[11] Donald Firesmith. Virtualization via virtual machines. https://insights.sei.cmu.edu/blog/virtualization-via-virtual-machines/, 2017.

[12] Prophesee. What is event based vision? https://www.prophesee.ai/2019/07/28/event-based-vision-2/, 2024.

[13] Alper Yilmaz, Omar Javed, and Mubarak Shah. Object tracking: A survey. *ACM computing surveys*, 38(2006-12), 2006.

[14] Zhe Chen, Zhibin Hong, and Dacheng Tao. An experimental survey on correlation filter-based tracking, 2015.

[15] Ziyuan Huang, Changhong Fu, Yiming Li, Fuling Lin, and Peng Lu. Learning aberrance repressed correlation filters for real-time uav tracking. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 2891–2900, 2019.

[16] David S. Bolme, J. Ross Beveridge, Bruce A. Draper, and Yui Man Lui. Visual object tracking using adaptive correlation filters. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 2544–2550, 2010.

[17] Xianfang Sun, Paul L. Rosin, Ralph R. Martin, and Frank C. Langbein. Bas-relief generation using adaptive histogram equalization. *IEEE Transactions on Visualization and Computer Graphics*, 15(4):642–653, 2009.

[18] Liyuan Li, Weimin Huang, Irene Yu-Hua Gu, and Qi Tian. Statistical modeling of complex backgrounds for foreground object detection. *IEEE Transactions on Image Processing*, 13(11):1459–1472, 2004.

[19] Guillermo Gallego, Tobi Delbruck, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J. Davison, Jorg Conradt, Kostas Daniilidis, and Davide Scaramuzza. Event-based vision: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(1):154–180, January 2022.

[20] Qinyi Wang, Yexin Zhang, Junsong Yuan, and Yilong Lu. Space-time event clouds for gesture recognition: From rgb cameras to event cameras. In *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 1826–1835, 2019.

[21] Cian Ryan, Brian O'Sullivan, Amr Elrasad, Aisling Cahill, Joe Lemley, Paul Kielty, Christoph Posch, and Etienne Perot. Real-time face & eye tracking and blink detection using event cameras. *Neural Networks*, 141:87–97, 2021.

[22] Iffatur Ridwan. *Looming Object Detection with Event-Based Cameras*. PhD thesis, 2018. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Last updated - 2023-03-04.

[23] Yijun Liu, Yuehai Chen, Wujian Ye, and Yu Gui. Fpga-nhap: A general fpga-based neuromorphic hardware acceleration platform with high speed and low power. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69:1–14, 06 2022.

[24] Amine Saddik, Rachid Latif, and Abdelhafid E. Ouardi. Low-power fpga architecture based monitoring applications in precision agriculture. *Journal of Low Power Electronics and Applications*, 11(4):39, 2021. Name - NVidia Corp; Copyright - © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2023-11-28; SubjectsTermNotLitGenreText - Morocco.

[25] DigiKey. Cyclone v fpga family. https://www.digikey.com.au/en/product-highlight/a/altera/cyclone-v-fpga-family, 2014.

[26] buildcomputers.net. Typical power consumption of pc components - power draw in watts. https://www.buildcomputers.net/power-consumption-of-pc-components.html, n.d.

[27] Julien Lamoureux and Steven J. E. Wilton. On the trade-off between power and flexibility of fpga clock networks. *ACM Trans. Reconfigurable Technol. Syst.*, 1(3), sep 2008.

[28] Yizhao Gao, Song Wang, and Hayden Kwok-Hay So. A reconfigurable architecture for real-time event-based multi-object tracking. *ACM Trans. Reconfigurable Technol. Syst.*, 16(4), sep 2023.

[29] Juan Barrios-Avilés, Taras Iakymchuk, Jorge Samaniego, Leandro D. Medus, and Alfredo Rosado-Muñoz. Movement detection with event-based cameras: Comparison with frame-based cameras in robot object tracking using powerlink communication. *Electronics*, 7(11):304, 2018. Copyright - © 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2023-11-23.

[30] Juan Barrios-Avilés, Alfredo Rosado-Muñoz, Leandro D. Medus, Manuel Bataller-Mompeán, and Juan F. Guerrero-Martínez. Less data same information for event-based sensors: A bioinspired filtering and data reduction algorithm. *Sensors*, 18(12), 2018.

[31] Gregory Kevin Cohen. *Event-Based Feature Detection, Recognition and Classification*. PhD thesis, The MARCS Institute Western Sydney University, 2015.

[32] Cedric Sheerlinck. *How to See with an Event Camera*. PhD thesis, The Australian National University, 2021.

[33] Prophesee. Prophesee and sony develop a stacked event-based vision sensor with the industry's smallest*1 pixels and highest*1 hdr performance. https://www.prophesee.ai/2020/02/19/prophesee-sony-stacked-event-based-vision-sensor/, 2020.

[34] Prophesee. *CCAM5 IMX636 TECHNICAL REFERENCE MANUAL*, April 2023.

[35] Sean Simmons. Latency in embedded systems. https://cs.uwaterloo.ca/~mkarsten/cs856-W10/lec06.pdf, 2009.

[36] ANSYS. What is an integrated circuit (ic)? https://www.ansys.com/blog/what-is-an-integrated-circuit, 2023.

[37] AMD. Downloads. https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html, n.d.

[38] Frank Vasquez and Chris Simmonds. *Mastering Embedded Linux Programming*. Packt Publishing, third edition, 2021.

[39] AMD. Downloads. https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/embedded-design-tools/archive.html, n.d.

[40] AMD. Yocto kria support. https://xilinx.github.io/kria-apps-docs/yocto/build/html/docs/yocto_kria_support.html, 2024.

[41] Xilinx. Build the petalinux image. https://xilinx.github.io/vmk180-trd/2020.2/platform1/html/build-plnx.html, 2020.

[42] Devsena Mishra. What is 'dependency hell'? https://medium.com/@devsenamishra/what-is-dependency-hell-b700fc937091, 2023.

[43] PROPHESEE Metavision Technologies. Metavision training videos — introduction to event-based vision sensor. https://www.youtube.com/watch?v=SPrdvhuAISk&ab_channel=PROPHESEEMetavisionTechnologies, 2023.

[44] Brian Benchoff. Bitbanging usb on low power arms. https://hackaday.com/2014/03/22/bitbanging-usb-on-low-power-arms/, 2014.

[45] Lachlan Spencer. honours_thesis_roi_data_collection_and_plotting. https://github.com/lachlanspencer/honours_thesis_roi_data_collection_and_plotting, 2024.