Fall 12-14-2024

Modeling the Performance and Resource Requirements for Gamma-Ray Telescope Signal Processing

Shijing Liang
Washington University – McKelvey School of Engineering

WASHINGTON UNIVERSITY IN ST. LOUIS

McKelvey School of Engineering
Department of Electrical & Systems Engineering

Modeling the Performance and Resource Requirements for Gamma-Ray Telescope Signal Processing

by
Shijing Liang

A thesis presented to
the McKelvey School of Engineering
of Washington University in
partial fulfillment of the
requirements for the degree
of Master of Science

December 2024
St. Louis, Missouri

Table of Contents

List of Figures

Acknowledgments

Throughout the process of completing this thesis, I received invaluable help and support from many people, to whom I am profoundly grateful.

First and foremost, I would like to express my deep appreciation to my advisor, Professor Chamberlain, whose unwavering guidance and support were pivotal throughout every stage of my research. From topic selection to the final write-up, he offered patient advice and encouragement, answering my questions with care and helping to calm my nerves during stressful moments. His guidance has been invaluable, and his support was as reassuring as family. I would also like to extend my heartfelt thanks to my PhD mentor, Marion Sudvarg, whose consistent and enthusiastic responses were an immense support. Marion always provided me with the information I needed, offering insightful feedback to my questions and guiding me toward new ideas when I felt stuck. Without the assistance of these two mentors, completing this thesis would not have been possible.

Additionally, I am grateful to my friends both in China and in the US. Every research journey has its highs and lows, and during the more challenging times, their encouragement and advice helped me regain my confidence. In the midst of writing pressures, their companionship brought me warmth and happiness.

I also want to thank my family, who have supported me steadfastly in the background, giving me their utmost encouragement and understanding. In moments of difficulty, they always believed I would make it to the end and complete this demanding task.

Finally, I am grateful to my university and lab for providing an excellent environment and ample resources that facilitated the successful completion of my research.

Once again, I extend my heartfelt thanks to everyone who has supported me along the way. This thesis is a testament to your wisdom and kindness. I hope my work can contribute, even modestly, to the realization of APT.

This work was supported by NASA award 80NSSC21K1741.

Shijing Liang

Washington University in St. Louis
December 2024

ABSTRACT OF THE THESIS

Modeling the Performance and Resource Requirements

for Gamma-Ray Telescope Signal Processing

by

Shijing Liang

Master of Science in Electrical Engineering

Washington University in St. Louis, 2024

Professor Roger D. Chamberlain, Chair

This thesis investigates the buffering requirements of the data pipeline for the Advanced Particle-astrophysics Telescope gamma-ray telescope. Given the importance and stochastic arrivals of astronomical signals, it is crucial to ensure that each gamma-ray signal is fully buffered to prevent data loss. To achieve this, FIFOs are inserted into the data processing pipeline to prevent bottlenecks during data packet processing.

Buffers play a critical role in regulating data flow, preventing data loss, and ensuring efficient data processing. They act as temporary storage areas, absorbing data surges and releasing it steadily, thus maintaining the pipeline's optimal performance. In general, buffer integration significantly enhances the stability and reliability of a data processing system.

The use of buffers ensures the integrity and timeliness of critical astronomical signals, providing reliable data support for subsequent investigation. This thesis uses discrete-event simulation models to assess the buffering requirements for a prototype gamma-ray telescope computational pipeline, ADAPT, the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope.

Chapter 1

Introduction

1.1 APT

The Advanced Particle-astrophysics Telescope (APT) is a planned space-based observatory designed for MeV to TeV gamma-ray astrophysics and cosmic-ray physics. APT's multimessenger capability relies on its ability to localize MeV transients in real time using on-board computational hardware. APT will combine a pair tracker and Compton telescope in a single monolithic design to include multiple layers of scintillating-fiber tracker hodoscopes and sodium-doped CsI scintillators. The CsI tiles are coupled with crossed planes of wavelength-shifting (WLS) fibers to localize energy deposition to ∼mm accuracy, as well as silicon photomultiplier (SiPM) based edge detectors to improve light collection and calorimetry. The mission concept [1] (see Figure 1.1), initial instrument simulations [2], and first version of the real-time gamma ray burst (GRB) reconstruction and localization algorithms [3] were presented at ICRC 2021.

1.2 ADAPT

1.2.1 Basic Information

The Antarctic Demonstrator for APT (ADAPT) is a prototype high-altitude balloon mission set to launch during the 2025–26 season. ADAPT aims to demonstrate APT's pair and Compton detection capabilities and serve as a proof of concept for prompt Compton reconstruction and localization, with the capability to send GRB positional alerts in real time. To evaluate ADAPT's performance, the team has developed a simulated model of the

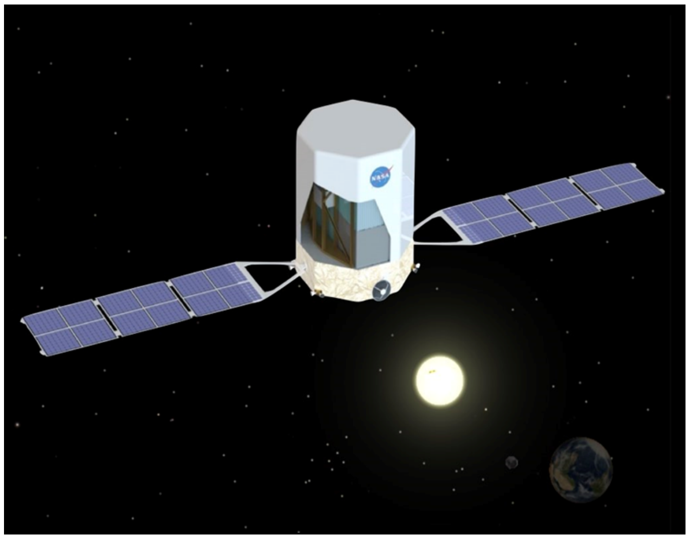
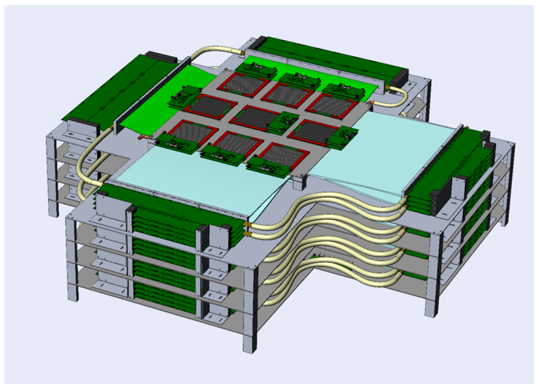


Figure 1.1: Advanced Particle-astrophysics Telescope (APT) [1].

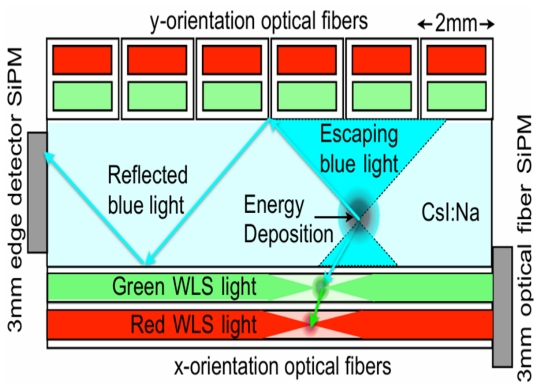
instrument that includes the optical properties of its CsI tiles, measurements of WLS signal attenuation, and characterizations of the SiPMs and preamplifier boards [4]. The model also accounts for additional uncertainties caused by the temporal effects of signal integration, including tail loss and event pileup.

1.2.2 Hardware Design Overview

The ADAPT instrument (illustrated in Figure 1.2(a)) has 4 primary detector layers, each comprised of an imaging CsI Calorimeter (ICC) and scintillating fiber tracker. Each ICC consists of a 3×3 arrangement of 5 mm thick, 15×15 cm sodium-doped CsI scintillating crystal tiles. Optical photons produced by energy deposits in the crystals are captured by perpendicular arrays of 2 mm wavelength-shifting (WLS) optical fibers running across the top and bottom surfaces of the CsI tiles and terminated at one end with 3×3 mm SiPMs; these allow precise localization of interactions in the crystals. To improve light detection and energy estimates, a detector that multiplexes 36 SiPMs is placed over each of the 6 tile edges on the adjacent ICC sides opposite the WLS fiber SiPMs. Below the ICC layers are 4 closely-spaced tail counters, which are identical to the ICCs but lack WLS fiber arrays, making them useful only for calorimetry. WLS fiber SiPM outputs are combined across a tile-width span (3-fiber multiplexing) into a SMART shaping preamplifier ASIC channel [5]. SMART outputs are sampled and digitized by a 16-channel ALPHA ASIC, a successor to the TARGET series [6]. The signals from all 36 SiPMs in an edge detector array are passively combined into both low- and high-gain preamplifier stages, each of which are also sampled by a dedicated ALPHA channel. The ALPHA ASIC captures 256 $\times$ 10 ns samples per channel in dual-banked analog memory, allowing simultaneous sampling and readout when triggered. Values are digitized and sent to an FPGA over a shared bus. With 225 WLS fibers running along each axis of an ICC layer, 5 ALPHA ASICs are sufficient to capture a single layer-axis. A sixth ALPHA will capture edge detector signals, with channels remaining open for other uses (e.g., capturing edge detectors from tail counter layers). A single Kintex-7 FPGA handles communication from the 6 ALPHAs for a layer-axis, then performs pedestal subtraction, signal integration, zero-suppression, and island detection. Both FPGAs send integrals to a third FPGA for each layer that performs centroiding and event building. Centroids are then sent to a CPU for backend data analysis, including Compton reconstruction and GRB localization [7].



(a) Hardware model



(b) Functional principle

Figure 1.2: Hardware design of Antarctic Demonstrator for APT (ADAPT) [8].

1.2.3 Motivation

The FPGAs that perform the front-end data processing (described in [9]) have limited on-chip memory, and there is concern on the part of the design team whether or not the on-chip memory will be sufficient for buffering of signals that come off the ALPHA chips, during processing, and prior to delivery to the downstream CPU.

This thesis investigates that concern through the use of discrete-event simulation models. The computational pipeline that executes on the FPGAs is modeled, with a focus primarily on the buffering requirements between computational stages.

We are able to both demonstrate the sufficiency of the on-chip memory for current buffering requirements as well as indicate the circumstances under which additional buffering will be required (e.g., higher ingest rates associated with the full APT instrument).

1.3 Thesis Structure and Contributions

Chapter 1: Introduction – Provide a concise overview of the project, outlining its primary objectives and significance. Describe the hardware design architecture for both the APT and ADAPT projects, detailing key components and configurations. Offer a brief summary of the thesis structure, explaining the organization of the chapters.

Chapter 2: Project Background – Present the background and context of the project, including a discussion of the data pipeline and its associated processes. Introduce the necessity of buffering within the pipeline, highlighting its role in maintaining data flow efficiency. Provide a brief literature review of relevant buffering techniques and research, setting the stage for further exploration.

Chapter 3: Methodology – Detail the research approach to studying buffering within the pipeline, including the step-by-step breakdown of the processes involved. Elaborate on the development of a simulation tool specifically designed to investigate buffering strategies within the pipeline. Discuss the parameters and conditions set within the simulation environment to ensure accurate and reliable results.

Chapter 4: Experimental Analysis – Present a comprehensive analysis of the experimental results obtained from various controlled studies using the simulator. Compare and contrast the outcomes under different conditions, highlighting the impact of varying buffering strategies on the overall system performance. Provide a detailed interpretation of the results, linking them to the initial research questions and hypotheses.

Chapter 5: Conclusion and Future Work – Summarize the key findings and conclusions drawn from the research, emphasizing the contributions to the field. Discuss the challenges encountered during the research process and their implications on the results. Offer insights into the potential future developments of the project, including suggestions for further research and possible applications of the findings.

Chapter 2

Background and Related Work

2.1 Data Pipeline

While ADAPT captures gamma rays, the signals it collects appear as peaks or“bumps” in the electrical signals from the SiPMs. To make these raw signals usable for further physical analysis, they must undergo several processing steps within a structured framework known as the data pipeline or the computational pipeline. This chapter focuses on data transmission and how the data is processed by different stages of the pipeline. Below is a simplified diagram illustrating the entire front-end design.

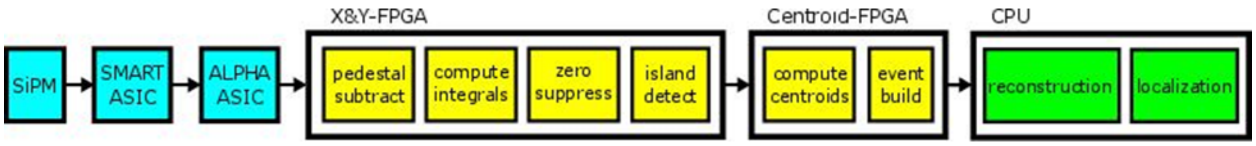


Figure 2.1: Computational pipeline for APT and ADAPT [8].

A single ALPHA data packet consists of an 8-word header, 4096 digitized sample words (16 channels $\times$ 256 samples), then a stop word. Of particular significance to the algorithms described below, the header includes (among other data) the following fields:

- Bank: Which of the two analog memory banks were read out.
- Fine_time: The sample number (current write position in the ring buffer) when the trigger arrives.
- Starting_sample: The number of the first sample listed in the output data packet.

In the current planned configuration (Figure 2.1), a single FPGA will perform pedestal subtraction, signal integration, zero-suppression and island detection across outputs from the 5 ALPHAs for a single layer-axis (labeled X-FPGA and Y-FPGA in the figure). Another FPGA (labeled Centroid-FPGA) will perform centroiding and event building across an entire layer, requiring a total of 12 FPGAs for ADAPT.

2.2 Data Process

The Compton computational pipeline is crafted to handle sensor data for astrophysical observations with high efficiency. It comprises a sequence of computational tasks that are executed by the FPGA-based front-end electronics, integral to preprocessing the sensor data into scientifically valuable information. The pipeline's key stages are illustrated in Figure 2.2.

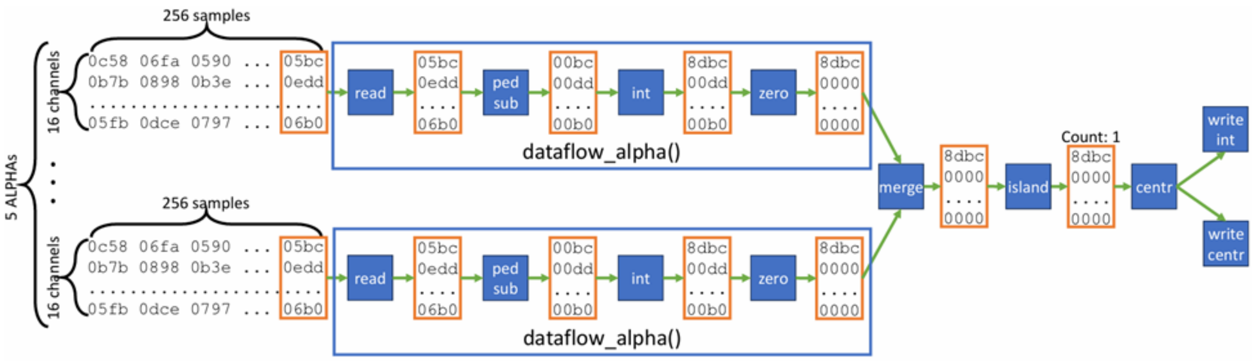


Figure 2.2: Dataflow architecture in detail [9].

There are seven stages for the FPGA data processing pipeline:

1. Read Data: The read process involves accessing raw data from multiple channels and samples, reading the values, and performing a pedestal subtraction to remove baseline noise.

2. Pedestal Subtraction: This step in the pipeline involves removing the baseline noise from the data, known as pedestal subtraction. This process is critical for isolating the true signal from the background noise inherent in the analog memory cells of the waveform 8 digitizers (ALPHA ASICs). The pedestal values are subtracted from the digitized readouts to obtain the actual signal values.

3. Signal Integration: Following pedestal subtraction, the signal integration stage sums up the signal over a specified window to quantify the energy captured by each pixel. This process involves integrating the waveform samples to infer the total number of photons detected, which is fundamental for determining the characteristics of the observed cosmic events.

4. Zero Suppression: To optimize data handling and storage, zero suppression is applied, which sets negligible signal values to zero. This stage reduces the volume of data that needs to be processed and transmitted, focusing on significant signal values that indicate actual astrophysical events.

5. Merge Integrals: The merge-integrals process combines the integrated outputs from multiple data sources, aggregates them, and centralizes the data for further processing. This ensures that the integrals from different data streams are merged into a single, unified dataset for subsequent steps.

6. Island Detection: The processed signals are then analyzed to identify clusters of adjacent pixels with non-zero values, termed islands. These islands represent potential astrophysical events or interactions within the detector. The identification of these islands is a crucial step in mapping the spatial distribution of the detected events.

7. Centroiding: The final stage in the FPGA pipeline is centroiding, where the center of gravity of the signal distribution within an island is calculated. This step provides a precise localization of the events, which is essential for reconstructing the direction and energy of the incident cosmic rays or gamma rays.

2.3 Considering Buffering Questions

Signals from the calorimeter fibers, edge-detectors, tail counters, and tracker fibers are continuously saved as an analog waveform on a series of switched-capacitor sample-and-hold circuits that operate as a ring buffer. Storing the signal in analog form requires dramatically less power than converting it to a digital signal. Upon receipt of a trigger, the triggered channel(s) undergo analog-to digital conversion (with 12-bit resolution) and communication to FPGAs for signal analysis. This technique has previously been deployed in terrestrial telescopes using the TARGET ASIC [6] and in particle physics experiments using the DRS4

ASIC chip [10]. For ADAPT, an ALPHA ASIC chip that performs buffering and A/D conversion is planned. The ALPHA supports 16 signal channels per chip. Given that the ALPHA chip is a new design, currently undergoing testing, we need a backup plan in the event that ALPHAs are not available in time for instrument fabrication. Alternative chips that will be considered should the ALPHA not be ready include the above-mentioned TARGET chip and its derivative HDSoC ASIC chip [11].

2.4 Related Work

Buffer occupancy is a critical concept in the design and optimization of data pipelines, particularly in systems with sequential data processing where different processors operate at varying speeds. Efficient buffer management is essential for preventing bottlenecks and ensuring smooth data flow between processors. This section reviews related work on buffer occupancy, discussing theoretical frameworks and real-world examples in various fields.

One of the key applications of buffer occupancy management is in network routers, where data packets must be queued before processing and forwarding. To prevent buffer overflow caused by varying input rates, techniques like Active Queue Management (AQM) [12], particularly Random Early Detection (RED) [13, 14], are employed. RED proactively drops packets before buffers fill up, reducing congestion and preventing data loss. This adaptive approach improves network performance by adjusting to fluctuating traffic, helping maintain smooth data transmission. Widely implemented in modern routers, RED and similar methods demonstrate the importance of managing buffer occupancy to minimize latency and ensure optimal system efficiency.

Additionally, in multicore processors, data is processed in parallel across cores with varying workloads and speeds, which can lead to bottlenecks if one core becomes overloaded. Buffers are used to temporarily store data between cores to prevent slow cores from stalling the entire processing pipeline. Research has shown that buffer occupancy is closely tied to workload distribution, and dynamic buffer management strategies have been developed to address this. For instance, Jiang et al. [15] demonstrated that by adjusting buffer sizes in real-time based on workload and processing speed, systems can optimize data flow and minimize delays. This approach is particularly useful in high-performance computing, where efficient data processing is crucial for maintaining maximum throughput.

Close to this thesis, in high-level synthesis (HLS)-based dataflow architectures, such as those used in FPGA implementations of graph neural networks (GNNs), buffer occupancy plays a critical role in managing task-level parallelism. The HLPerf framework, introduced by Zhao et al. [16], demonstrates how First In, First Out (FIFO) buffers act as temporary storage between processing stages, ensuring smooth data flow. The framework simulates GNN models and graph datasets to assess how buffer sizes impact system throughput. Small buffers can cause data stalls, while oversized buffers may waste resources without performance gains. HLPerf shows that optimal buffer sizes depend on the specific GNN model and dataset, highlighting the importance of buffer management in optimizing performance in FPGA-based dataflow systems. Similarly, Faber and Chamberlain [17] model the BLAST [18, 19] computational biology application with both simulation techniques and network calculus [20, 21].

Simulation modeling has been widely used in the literature to assess buffering requirements and the performance of pipelined systems. Kleinrock's work on queueing theory established early models for analyzing buffer occupancy in systems with variable processing times [22]. Jiang et al. introduced dynamic buffer resizing in multicore processors, optimizing real-time data flow between cores [15]. Additionally, El Meligy et al. applied simulation-based methods in adaptive bitrate streaming to optimize video quality by managing buffer occupancy based on fluctuating network conditions [23]. Similarly, HLPerf by Zhao et al. used simulation to evaluate the performance of dataflow architectures in FPGA-based systems, focusing on the impact of buffer sizes in Graph Neural Networks (GNNs) [16]. These studies underscore the importance of simulation in optimizing buffer management for enhanced system performance. More broadly, simulation (in particular, discrete-event simulation) has been used to model a wide variety of systems [24, 25, 26].

Chapter 3

Simulator Design

To determine how buffer occupancy impacts the effectiveness of our data pipeline, we designed a discrete-event simulator to model the data flow through all of the pipeline stages. This chapter outlines the construction of this simulator.

3.1 Simulator Module Simplified

The pipeline involves five ALPHA ASICs for detection, each group delivering data to an FPGA for x-axis or y-axis processing, as well as an FPGA for centroiding. As discussed in Chapter 2, each group of five ALPHA ASICs sequentially transfers data to both the X-FPGA and Y-FPGA. Following zero suppression, a merging step occurs. This necessitates a loop in the X-FPGA to perform the initial three steps: pedestal subtraction, signal integration, and zero suppression. The Y-FPGA follows the same procedure. For simplicity in the simulator design, we model the data flow as a single loop through the pipeline. Once this single loop is functioning correctly, we incorporate the remaining four loops into the simulator. The resulting pipeline, with buffers positioned between each processing stage, is illustrated in Figure 3.1.

A raw data packet containing 65,680 bits is read and transferred to the Pedestal Subtraction stage. The size remains unchanged until it passes through Compute Integrals, reducing to 2,192 bits. After Merging Integrals, the package size increases to 10,960 bits, and further to 11,600 bits after the Island Detection processor. Ultimately, at the end of the pipeline, the data package size reaches 17,360 bits, which is the final size for export from the processing procedure to the network interface. The packet sizes are illustrated in Figure 3.2. Note that these packet sizes, and the subsequent initiation intervals discussed next, are those

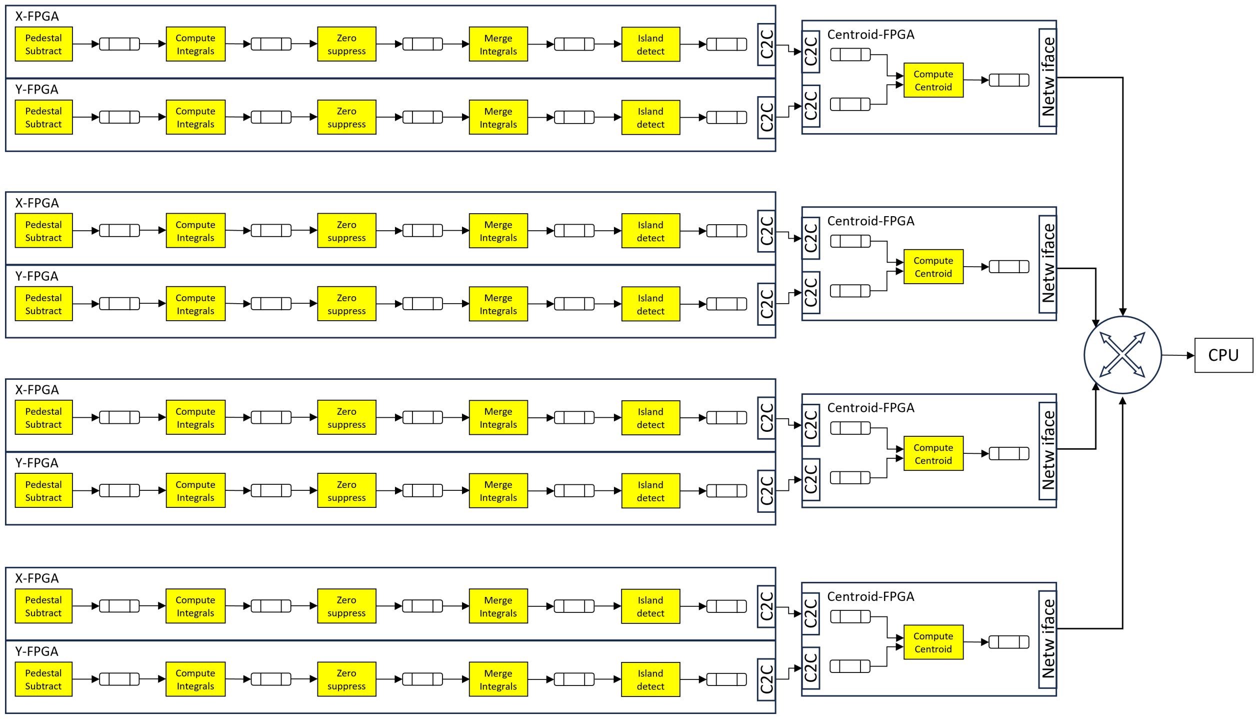


Figure 3.1: Computational pipeline for all four detector layers.

documented in Sudvarg et al. [9] and are straightforward to alter within the simulation model.

In the overall processing workflow, there are different processing initiation intervals for each pipeline stage. The processing cycles for the seven processors are as follows: reading data requires 267 clock cycles, pedestal subtraction requires 258 clock cycles, computing integrals requires 273 clock cycles, zero suppression requires 6 clock cycles, merging integrals requires 22 clock cycles, island detection requires 105 clock cycles, and computing centroid requires 137 clock cycles [9].

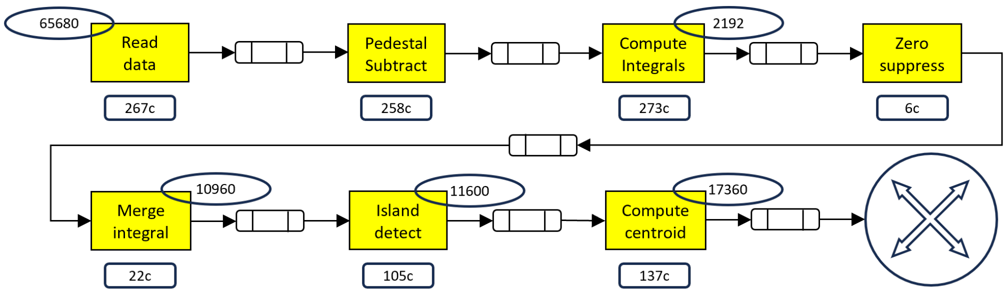


Figure 3.2: Simplified computational pipeline.

The initial analysis in [9] neglected the chip-to-chip interface (labeled C2C in Figure 3.1) between the X and Y FPGAs and the Centroid FPGA (i.e., the analysis corresponded to the diagram of Figure 3.2). While our initial discrete-event simulation model followed this practice, we subsequently expanded the simulator to explicitly include the chip-to-chip interface, as illustrated in Figure 3.3.

In the transition from Figure 3.2 to Figure 3.3, the Chip-to-Chip (C2C) Channel is introduced, providing a critical enhancement in data transfer between distinct processing units. This channel enables efficient inter-chip communication, improving the accuracy of the processing pipeline model. Specifically, after the island detection phase, data is transmitted across chips via the C2C channel at a rate of approximately 50 Mbps. This addition facilitates the separation of computational tasks across different hardware modules, allowing for better load distribution and parallelism, and more closely models the actual hardware design. In the following chapter, we will investigate the implications of varying data rates in the chip-to-chip channel.

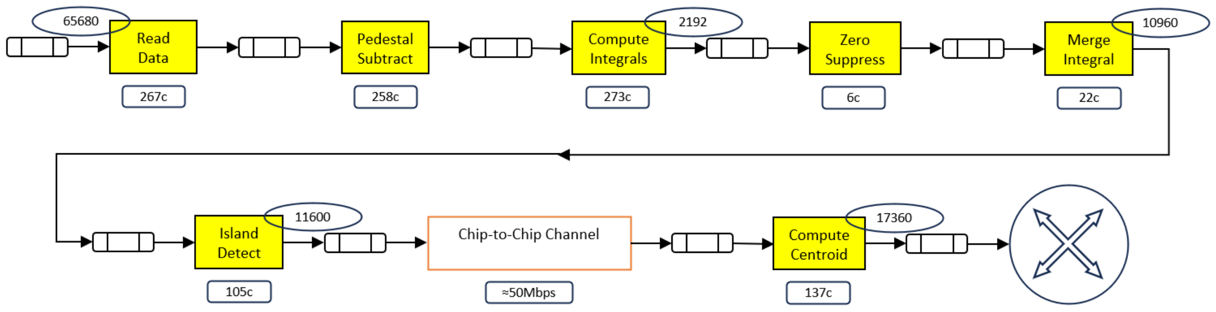


Figure 3.3: Simplified computational pipeline including chip-to-chip interface.

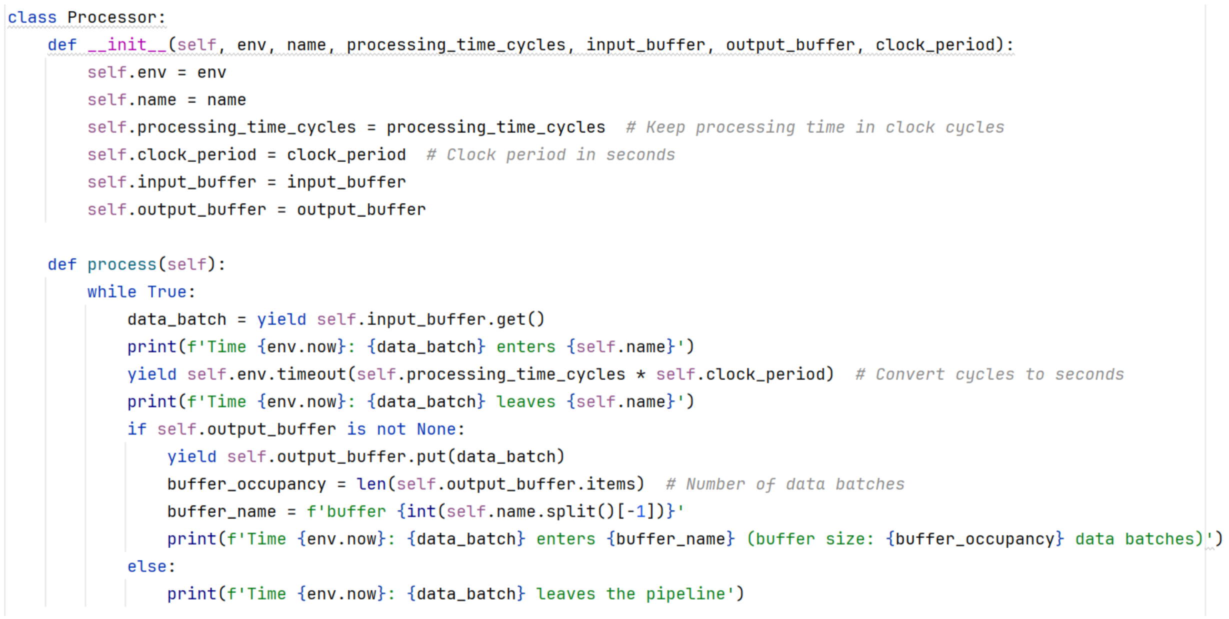
3.2 Simulation Code Design

In designing the overall simulator, we first established the most basic pipeline structure. During this design process, we retained only the initiation intervals of each processor. Other fundamental parameters, such as input rate, clock cycle, and output rate, were intentionally disregarded to ensure that the simulator could handle data packets in parallel processing mode. This approach prevents the pipeline from operating in a serial mode, where the next data packet would only be processed after the current one has fully traversed all seven processors. This section outlines the construction and execution of a data pipeline simulation using the SimPy library [27]. The simulation models a series of processors and buffers to evaluate the performance and efficiency of data processing workflows. It is authored in the process style [26], supported by SimPy, as opposed to the event-driven style. This allows individual modeled elements to be expressed separately from one another, and the simulation execution to proceed as a set of co-routines.

3.2.1 Process

The process method in this code simulates the flow of data batches through a processing stage within a data pipeline. This method (shown in Figure 3.4) continuously processes incoming data batches in a loop. Each data batch is retrieved from an `input_buffer`, and the current simulation time and data batch details are recorded when it enters the processing stage. The state of the corresponding output buffer, identified by the last character of the processor's name, is also recorded, showing the number of packets it contains.

The method then simulates the processing time by pausing the execution for a specified duration (`self.processing_time`). After this period, the data batch exits the processing stage, and a record is printed indicating this event along with the current simulation time. The data batch is then transferred to the `output_buffer`, and the updated state of this buffer is recorded. This process repeats indefinitely, simulating a continuous data flow through the pipeline.



```
class Processor:
    def __init__(self, env, name, processing_time_cycles, input_buffer, output_buffer, clock_period):
        self.env = env
        self.name = name
        self.processing_time_cycles = processing_time_cycles  # Keep processing time in clock cycles
        self.clock_period = clock_period  # Clock period in seconds
        self.input_buffer = input_buffer
        self.output_buffer = output_buffer

    def process(self):
        while True:
            data_batch = yield self.input_buffer.get()
            print(f'Time {env.now}: {data_batch} enters {self.name}')
            yield self.env.timeout(self.processing_time_cycles * self.clock_period)  # Convert cycles to seconds
            print(f'Time {env.now}: {data_batch} leaves {self.name}')
            if self.output_buffer is not None:
                yield self.output_buffer.put(data_batch)
                buffer_occupancy = len(self.output_buffer.items)  # Number of data batches
                buffer_name = f'buffer {int(self.name.split()[-1])}'
                print(f'Time {env.now}: {data_batch} enters {buffer_name} (buffer size: {buffer_occupancy} data batches)')
            else:
                print(f'Time {env.now}: {data_batch} leaves the pipeline')
```

Figure 3.4: Processing procedure definition.

3.2.2 Network Interface

The `transmit` function (Figure 3.5) models the transmission of data batches between buffers in the pipeline that are data rate-limited (e.g., the network link from the FPGA to the CPU). Operating in a continuous loop, it retrieves a data batch from the input buffer and calculates the transmission time based on the data size and output rate. The function waits for the calculated transmission time to simulate the transfer process. Afterward, it logs the departure of the data batch from the processor and, if an output buffer is available, places the data batch into it, logging the buffer's occupancy. This function ensures accurate modeling of data transmission times and buffer management within the pipeline simulation.

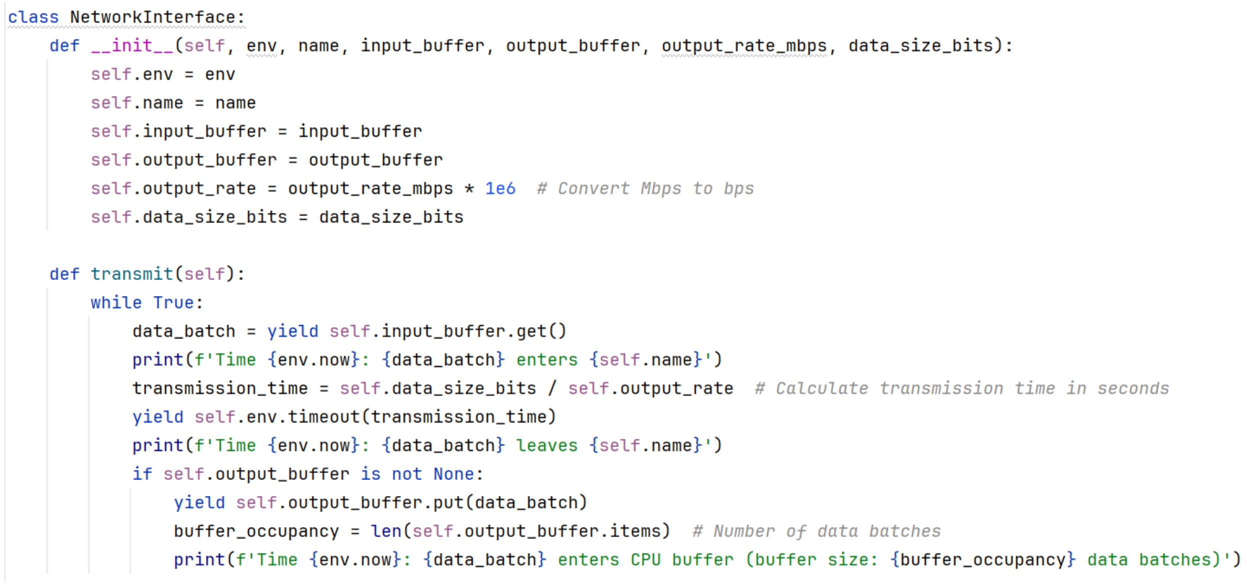
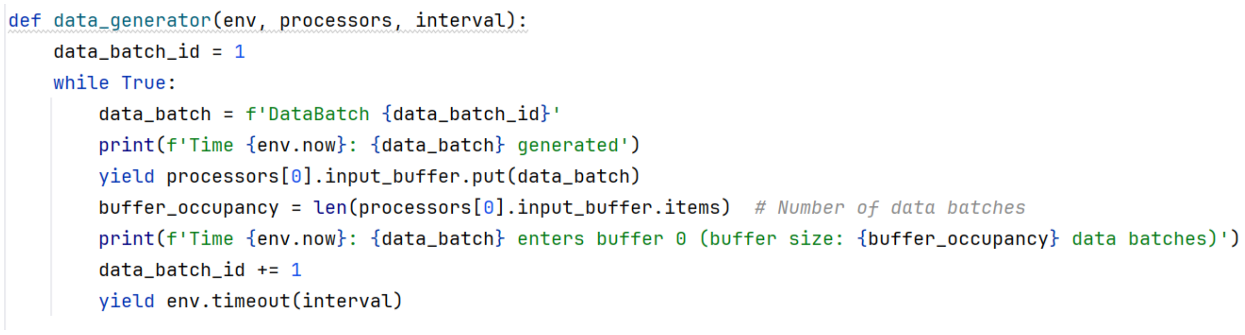

```python
class NetworkInterface:
    def __init__(self, env, name, input_buffer, output_buffer, output_rate_mbps, data_size_bits):
        self.env = env
        self.name = name
        self.input_buffer = input_buffer
        self.output_buffer = output_buffer
        self.output_rate = output_rate_mbps * 1e6  # Convert Mbps to bps
        self.data_size_bits = data_size_bits

    def transmit(self):
        while True:
            data_batch = yield self.input_buffer.get()
            print(f'Time {env.now}: {data_batch} enters {self.name}')
            transmission_time = self.data_size_bits / self.output_rate  # Calculate transmission time in seconds
            yield self.env.timeout(transmission_time)
            print(f'Time {env.now}: {data_batch} leaves {self.name}')
            if self.output_buffer is not None:
                yield self.output_buffer.put(data_batch)
                buffer_occupancy = len(self.output_buffer.items)  # Number of data batches
                print(f'Time {env.now}: {data_batch} enters CPU buffer (buffer size: {buffer_occupancy} data batches)')
```

Figure 3.5: Network interface definition.

3.2.3 Data Generator

The `data_generator` function (Figure 3.6) simulates the creation and introduction of data batches into the processing pipeline at random intervals. Each data batch is uniquely identified by an incrementing `data_batch_id`. Within the function, a data batch is generated and labeled as DataBatch {`data_batch_id`}, and then placed into the input buffer of the first processor. This insertion is followed by log statements that record the current simulation time and the status of the buffer, providing insight into the data flow through the pipeline.

The function continuously generates data batches, with each batch entering the first processor's input buffer. The `yield env.timeout(200)` statement introduces a fixed delay of 200 simulation time units between the generation of consecutive data batches, simulating a deterministic input data rate. This is altered in some experiments to a Poisson arrival process (exponentially distributed interarrival times). The random interval ensures variability in the arrival of data batches, which better mimics real-world data processing scenarios and tests the pipeline's ability to handle data under fluctuating conditions.

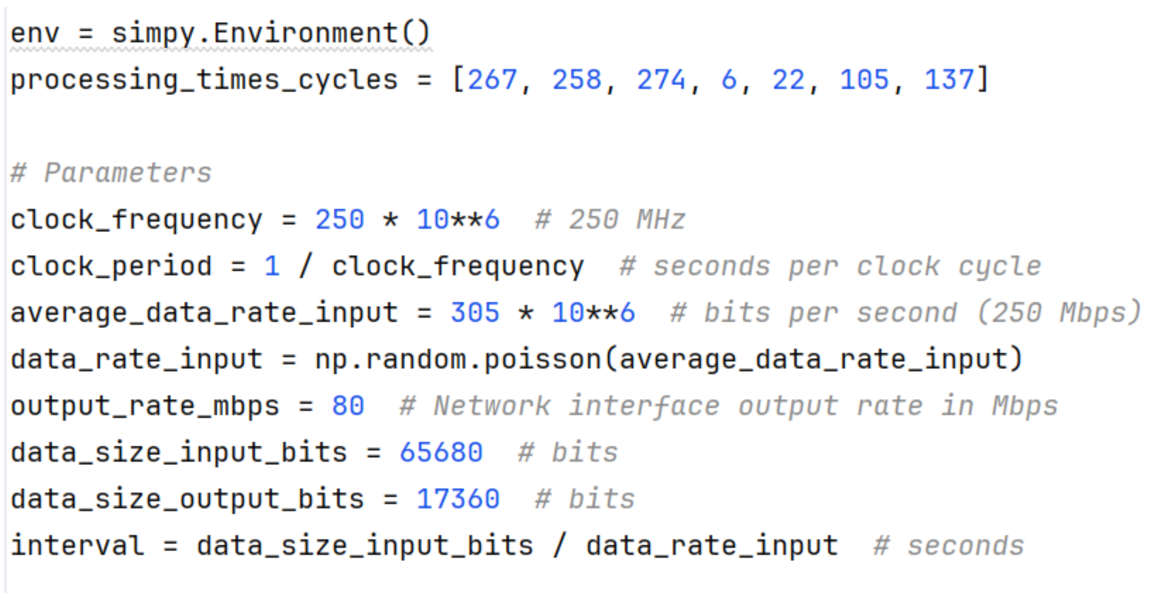


```python
def data_generator(env, processors, interval):
    data_batch_id = 1
    while True:
        data_batch = f'DataBatch {data_batch_id}'
        print(f'Time {env.now}: {data_batch} generated')
        yield processors[0].input_buffer.put(data_batch)
        buffer_occupancy = len(processors[0].input_buffer.items)  # Number of data batches
        print(f'Time {env.now}: {data_batch} enters buffer 0 (buffer size: {buffer_occupancy} data batches)')
        data_batch_id += 1
        yield env.timeout(interval)
```

Figure 3.6: Data generator definition.

3.2.4 Parameter Setting

The code in Figure 3.7 initializes a simulation environment to model a data pipeline, focusing on processing times and data flow. The environment is set up using SimPy, with processing times defined for each stage in clock cycles. Key parameters include a clock frequency of 250 MHz, input data rate of 250 Mbps, output rate of 66 Mbps, and data sizes for input and output. The interval for data packet generation is calculated based on the input data size and rate. Buffers are created to store data between stages, and processors are instantiated with their respective processing times and clock period. This setup simulates the parallel processing of data packets through multiple stages, allowing for the analysis of pipeline performance and efficiency.

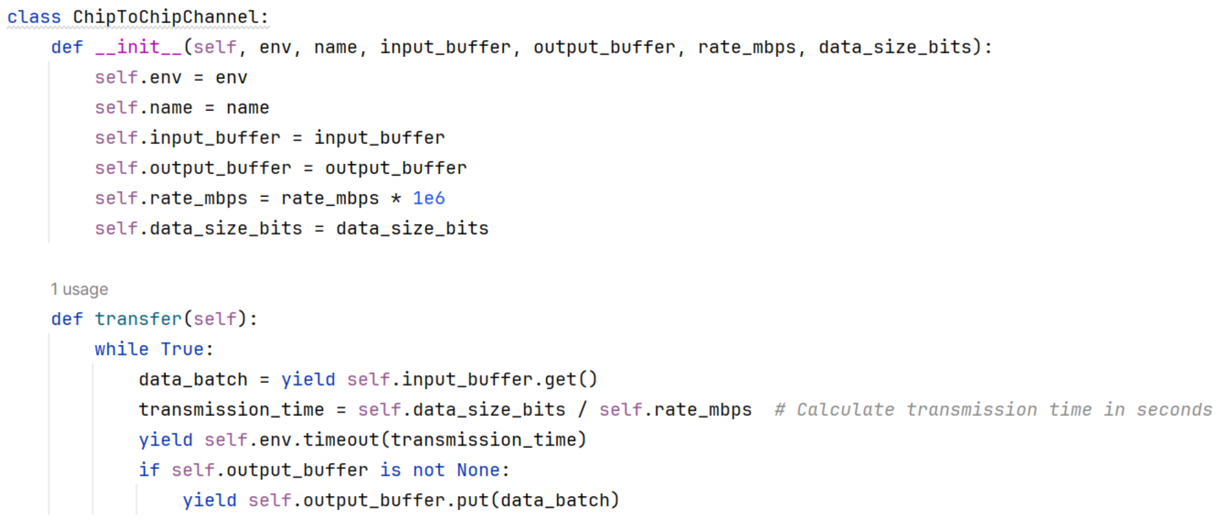


```python
env = simpy.Environment()
processing_times_cycles = [267, 258, 274, 6, 22, 105, 137]

# Parameters
clock_frequency = 250 * 10**6  # 250 MHz
clock_period = 1 / clock_frequency  # seconds per clock cycle
average_data_rate_input = 305 * 10**6  # bits per second (250 Mbps)
data_rate_input = np.random.poisson(average_data_rate_input)
output_rate_mbps = 80  # Network interface output rate in Mbps
data_size_input_bits = 65680  # bits
data_size_output_bits = 17360  # bits
interval = data_size_input_bits / data_rate_input  # seconds
```

Figure 3.7: Initial parameter settings.

3.2.5 Chip-to-chip Channel

The chip-to-chip code of Figure 3.8 initializes a simulation environment to model a data pipeline, focusing on processing times and data flow. The interval for data packet generation is calculated based on the input data size and rate, similar to the network model.



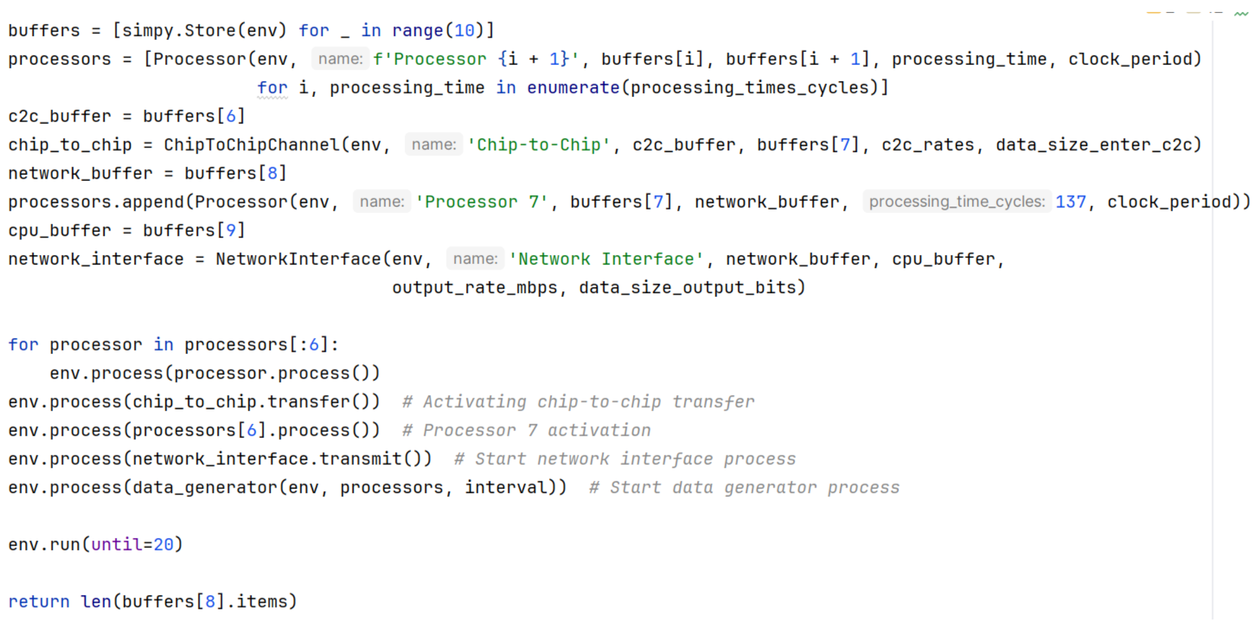
```python
class ChipToChipChannel:
    def __init__(self, env, name, input_buffer, output_buffer, rate_mbps, data_size_bits):
        self.env = env
        self.name = name
        self.input_buffer = input_buffer
        self.output_buffer = output_buffer
        self.rate_mbps = rate_mbps * 1e6
        self.data_size_bits = data_size_bits

    1 usage
    def transfer(self):
        while True:
            data_batch = yield self.input_buffer.get()
            transmission_time = self.data_size_bits / self.rate_mbps  # Calculate transmission time in seconds
            yield self.env.timeout(transmission_time)
            if self.output_buffer is not None:
                yield self.output_buffer.put(data_batch)
```

Figure 3.8: Chip-to-chip definition.

3.2.6 Simulator

The code of Figure 3.9 stitches the above elements together and simulates a hardware or network communication system using SimPy, a discrete-event simulation library. It models multiple processors, buffers, and communication channels interacting within the environment (`env`). Buffers are represented by `simPy.Store` objects, where each processor is associated with a buffer for queuing data during processing. Seven processors are initialized with varying processing times, simulating a realistic processing system.

The simulation also models communication between processors through a Chip-To-Chip Channel and a Network Interface, which facilitate data transfers between components. Processes for each processor, the chip-to-chip transfer, and the network interface are activated, allowing for a detailed simulation of the system's behavior over time.



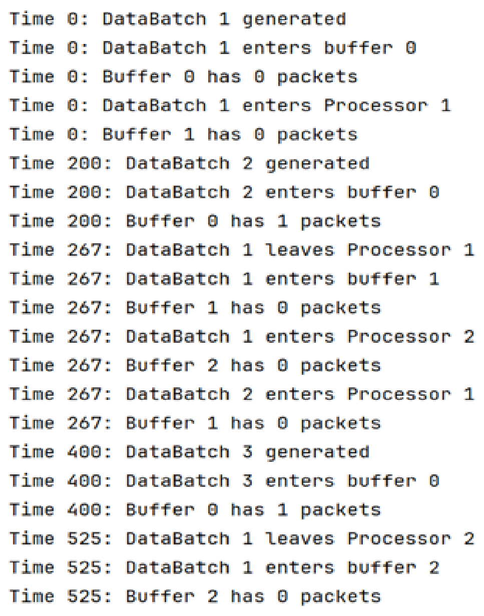
```
buffers = [simpy.Store(env) for _ in range(10)]
processors = [Processor(env,  name: f'Processor {i + 1}', buffers[i], buffers[i + 1], processing_time, clock_period)
                        for i, processing_time in enumerate(processing_times_cycles)]
c2c_buffer = buffers[6]
chip_to_chip = ChipToChipChannel(env,  name: 'Chip-to-Chip', c2c_buffer, buffers[7], c2c_rates, data_size_enter_c2c)
network_buffer = buffers[8]
processors.append(Processor(env,  name: 'Processor 7', buffers[7], network_buffer,  processing_time_cycles: 137, clock_period))
cpu_buffer = buffers[9]
network_interface = NetworkInterface(env,  name: 'Network Interface', network_buffer, cpu_buffer,
                                     output_rate_mbps, data_size_output_bits)

for processor in processors[:6]:
    env.process(processor.process())
env.process(chip_to_chip.transfer())  # Activating chip-to-chip transfer
env.process(processors[6].process())  # Processor 7 activation
env.process(network_interface.transmit())  # Start network interface process
env.process(data_generator(env, processors, interval))  # Start data generator process

env.run(until=20)

return len(buffers[8].items)
```

Figure 3.9: Simulator design.

3.3 Initial Testing Results

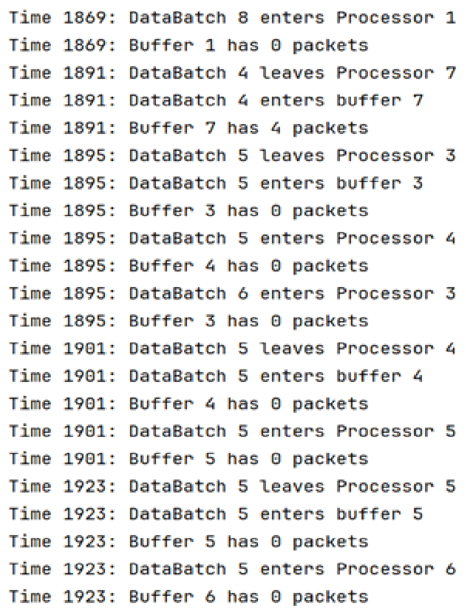
3.3.1 Basic Code Running Result

While running the basic pipeline code, we get the results shown in Figure 3.10.



```
Time 0: DataBatch 1 generated
Time 0: DataBatch 1 enters buffer 0
Time 0: Buffer 0 has 0 packets
Time 0: DataBatch 1 enters Processor 1
Time 0: Buffer 1 has 0 packets
Time 200: DataBatch 2 generated
Time 200: DataBatch 2 enters buffer 0
Time 200: Buffer 0 has 1 packets
Time 267: DataBatch 1 leaves Processor 1
Time 267: DataBatch 1 enters buffer 1
Time 267: Buffer 1 has 0 packets
Time 267: DataBatch 1 enters Processor 2
Time 267: Buffer 2 has 0 packets
Time 267: DataBatch 2 enters Processor 1
Time 267: Buffer 1 has 0 packets
Time 400: DataBatch 3 generated
Time 400: DataBatch 3 enters buffer 0
Time 400: Buffer 0 has 1 packets
Time 525: DataBatch 1 leaves Processor 2
Time 525: DataBatch 1 enters buffer 2
Time 525: Buffer 2 has 0 packets
```

(a)Testing Result(1)



```
Time 1869: DataBatch 8 enters Processor 1
Time 1869: Buffer 1 has 0 packets
Time 1891: DataBatch 4 leaves Processor 7
Time 1891: DataBatch 4 enters buffer 7
Time 1891: Buffer 7 has 4 packets
Time 1895: DataBatch 5 leaves Processor 3
Time 1895: DataBatch 5 enters buffer 3
Time 1895: Buffer 3 has 0 packets
Time 1895: DataBatch 5 enters Processor 4
Time 1895: Buffer 4 has 0 packets
Time 1895: DataBatch 6 enters Processor 3
Time 1895: Buffer 3 has 0 packets
Time 1901: DataBatch 5 leaves Processor 4
Time 1901: DataBatch 5 enters buffer 4
Time 1901: Buffer 4 has 0 packets
Time 1901: DataBatch 5 enters Processor 5
Time 1901: Buffer 5 has 0 packets
Time 1923: DataBatch 5 leaves Processor 5
Time 1923: DataBatch 5 enters buffer 5
Time 1923: Buffer 5 has 0 packets
Time 1923: DataBatch 5 enters Processor 6
Time 1923: Buffer 6 has 0 packets
```

(b)Testing Result(2)

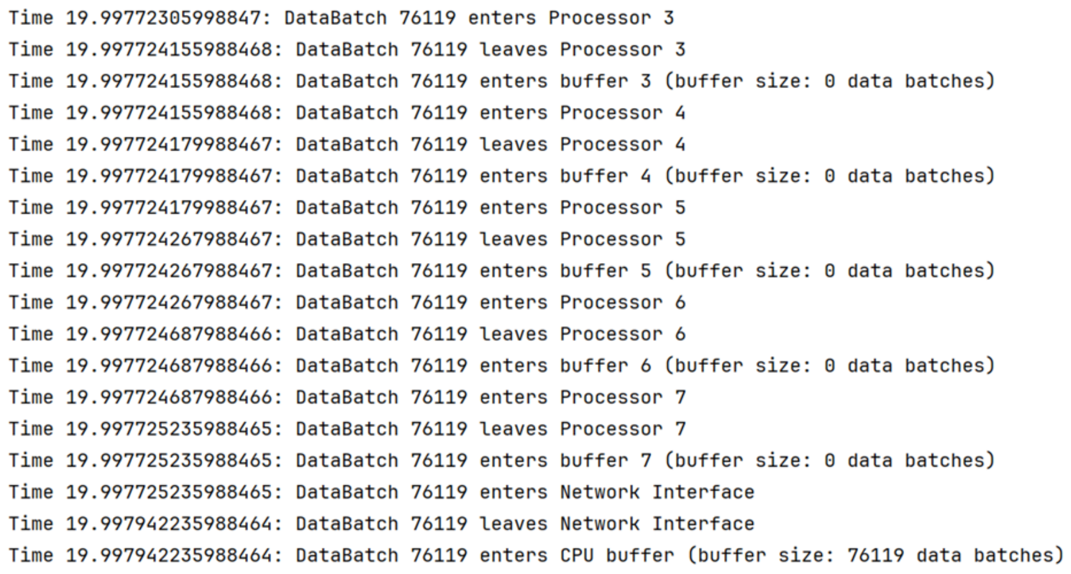
Figure 3.10: Testing results samples.

From the output results, we can tell when a data package is generated, when does it enter the data pipeline, and also for the exact time when it enters and exits from a buffer. We named the buffer with its order in the pipeline, and adding a buffer 0 in the very front of the pipeline to see if there is any data package being clogged even before it is input to the pipeline.

This test gives us some confidence that the pipeline is working with data packages in parallel. In this case, we can continue by adding parameters: input rate, output rate, and clock.

3.3.2 Buffer Occupancy Code Running Result

To make sure that the data package is not stuck inside the pipeline, we added a buffer after the network interface buffer. This is set to represent how many data packages are successfully exported from the pipeline. Figure 3.11 is an example of the output results. We can tell from the output file there is an exact timetable of the status of each data package.



```
Time 19.99772305998847: DataBatch 76119 enters Processor 3
Time 19.997724155988468: DataBatch 76119 leaves Processor 3
Time 19.997724155988468: DataBatch 76119 enters buffer 3 (buffer size: 0 data batches)
Time 19.997724155988468: DataBatch 76119 enters Processor 4
Time 19.997724179988467: DataBatch 76119 leaves Processor 4
Time 19.997724179988467: DataBatch 76119 enters buffer 4 (buffer size: 0 data batches)
Time 19.997724179988467: DataBatch 76119 enters Processor 5
Time 19.997724267988467: DataBatch 76119 leaves Processor 5
Time 19.997724267988467: DataBatch 76119 enters buffer 5 (buffer size: 0 data batches)
Time 19.997724267988467: DataBatch 76119 enters Processor 6
Time 19.997724687988466: DataBatch 76119 leaves Processor 6
Time 19.997724687988466: DataBatch 76119 enters buffer 6 (buffer size: 0 data batches)
Time 19.997724687988466: DataBatch 76119 enters Processor 7
Time 19.997725235988465: DataBatch 76119 leaves Processor 7
Time 19.997725235988465: DataBatch 76119 enters buffer 7 (buffer size: 0 data batches)
Time 19.997725235988465: DataBatch 76119 enters Network Interface
Time 19.997942235988464: DataBatch 76119 leaves Network Interface
Time 19.997942235988464: DataBatch 76119 enters CPU buffer (buffer size: 76119 data batches)
```

Figure 3.11: Buffer occupancy with time frame.

Chapter 4

Results

We developed the discrete-event simulator for the pipeline not only to determine the critical point at which buffering is necessary but also to understand the buffer capacity required within the pipeline. It is crucial to note that while the buffer capacity can be adjusted, it is not infinite and cannot be modified arbitrarily during data processing. Therefore, we must establish a specific buffer capacity value before the pipeline operation begins. This raises a critical question: how can we determine the maximum buffer size needed, referred to as the maximum occupancy?

During the execution of the basic simulation code, we observed that regardless of the variations in input and output rates, the buffer occupancy within the pipeline remained empty, except for the buffer at the end of the pipeline, right before the network interface, which we refer to as the network buffer. This buffer exhibited a transition from empty to occupied as data completed processing and was exported to the network interface. Consequently, our initial study shifts focus from analyzing all buffers within the pipeline to specifically investigating the occupancy of the network buffer. This forms the initial objective of this chapter. It is still important to retain the preceding buffers to maintain the pipeline's integrity and we subsequently investigate occupancy in intermediate buffers that will arise due to parameter variations from those initially considered.

4.1 Fixed Input and Output Rates

Based on the maximum data rate supported by the ALPHA chips, data enters the computational pipeline at an input rate of approximately 250 Mbps. Considering the operational status of the network interface, we have estimated the output rate to be 80 Mbps for initial simulation modeling. The simulation is illustrated in Figure 4.1 and the simulation results are shown in Figure 3.11. As can be seen, during the data transmission process, the time interval between incoming data packets is significantly longer than the time it takes for a packet to traverse the entire pipeline. Consequently, there is no need for buffers during the system's operation other than a single latch for continuity. Regardless of the number of data packets transmitted, the buffer occupancy between any two processors remains zero. This is because the processed data packet can immediately proceed to the next processor without waiting in the buffer.

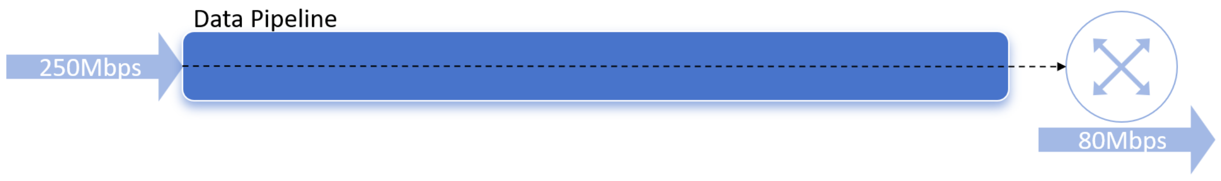


Figure 4.1: Simplified data pipeline with fixed input and output.

Given these conditions, we are reassured that, under the specified parameters, the data transmission process remains smooth, preventing pipeline congestion as long as the processors handle data efficiently. This insight leads us to the next line of inquiry: by adjusting one or more of the parameters, can we identify a circumstance or set of circumstances were buffering is required? At this point, the presence of buffers would become significant for temporarily storing unprocessed data packets.

4.2 Varying Input Data Rate

To identify a critical point for the input data rate, we set the output rate to a constant 80 Mbps, the clock to 250 MHz, the runtime to 20 seconds, and varied the input data rate. Through continuous adjustments, we discovered that with a fixed output rate of 80 Mbps, the input rate could be increased to approximately 300 Mbps without increasing the buffer occupancy. To better visualize the buffer occupancy at various input rates, we can refer to Figure 4.2.

Figure 4.3 plots the input rate on the x-axis and buffer occupancy on the y-axis. It reveals that when the input rate is less than or equal to 302 Mbps, the network buffer consistently shows zero occupancy, indicating that no data packets are retained and the pipeline remains unimpeded. However, once the input rate exceeds 303 Mbps, the network buffer begins to

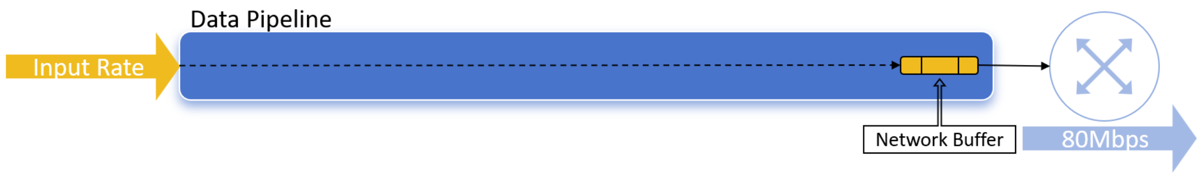


Figure 4.2: Simplified data pipeline with fixed output.

accumulate data packets. Thus, under these conditions, 302 Mbps is identified as the critical (threshold) input rate.

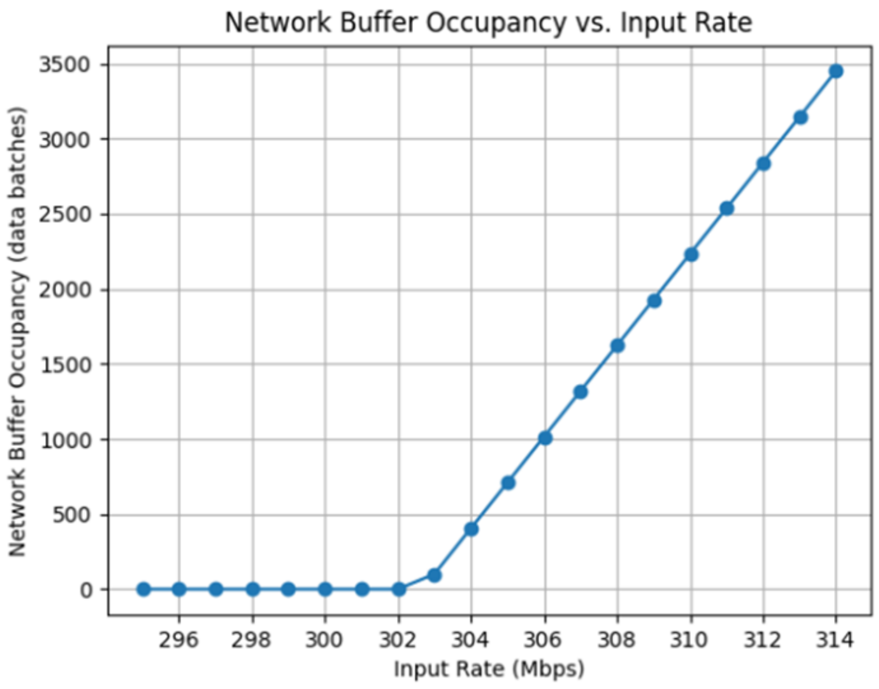


Figure 4.3: Buffer occupancy with ranging input rate (setting output rate to be 80 Mbps).

4.3 Varying Output Rate

We seek to identify another critical point: with the input rate fixed at 250 Mbps, and maintaining the clock at 250 MHz and the runtime at 20 seconds, we will observe the corresponding impact of varying the output data rate. By systematically analyzing the behavior of the system under these conditions, we aim to determine the minimum output rate needed to avoid buffer occupancy and ensure smooth data flow through the pipeline.

This analysis will provide further insights into the relationship between input and output rates and help optimize the system's performance.

This experiment is illustrated in Figure 4.4 and the simulation results (showing the buffer occupancy of the network interface buffer at various output rates) are plotted in Figure 4.5.

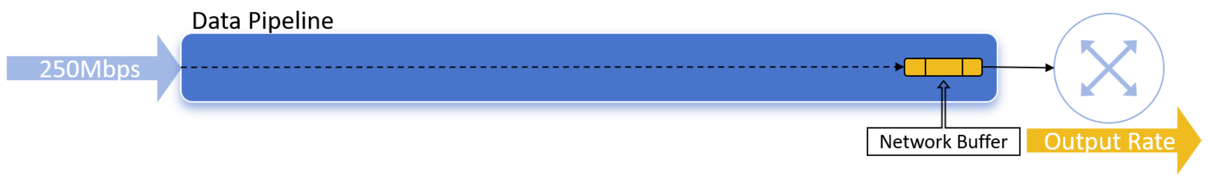


Figure 4.4: Simplified data pipeline with fixed input.

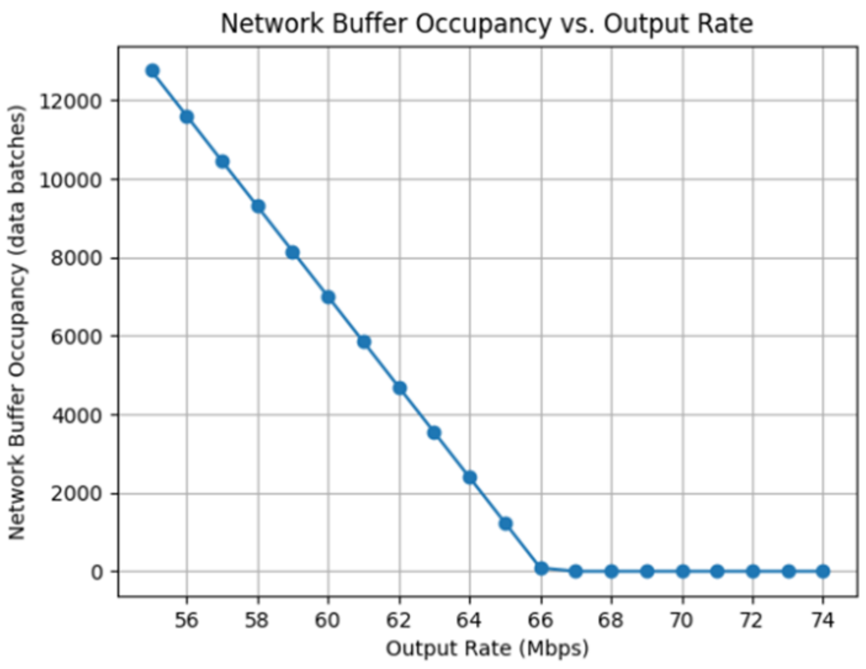


Figure 4.5: Buffer occupancy with ranging output rate (setting input rate to be 250 Mbps).

Given the fact that the regularity of execution times in the pipeline implies that significant buffering only happens in the network buffer, it is not surprising that the ratio of input data rate to output data rate is the same at the critical point for both Figures 4.3 and 4.5.

4.4 Varying Execution Time

After completing the control variable study of input and output rates, we found that the pipeline's runtime also affects the network interface buffer's occupancy. This is understandable because when the output rate is significantly lower than the input rate, the network interface buffer's occupancy gradually increases over time. The larger the difference between input and output rates, the faster the occupancy of the buffer increases.

By plotting curves for different runtimes and input rates, while keeping the output rate fixed at 80 Mbps, we observe that when the input rate is controlled at or below 302 Mbps, the buffer occupancy remains zero regardless of the runtime. This is because the input and output rates are balanced under these conditions. When the input rate increases to 303 Mbps, the network interface buffer shows non-zero occupancy, but the increase is not significant enough to impact the pipeline's operation severely, allowing us to maintain buffer capacity within a fixed range to ensure smooth pipeline operation.

However, as the input rate continues to rise, the buffer's occupancy varies significantly with different runtimes, which can severely disrupt the pipeline's normal operation, leading to either data loss or severe data flow congestion. Therefore, through simulation testing using this simulator, we can determine input and output rates that are feasible and conducive to maintaining a stable and efficient pipeline.

These results are illustrated in Figure 4.6, which plots network interface buffer occupancy as a function of input rate for a variety of runtimes. As expected, above the input rate of $\sim$302 Mbps, the occupancy increases linearly with runtime. Below this input rate, the network buffer stays empty.

Similarly, assuming we fix the input rate at a constant 250 Mbps and adjust the output rate and runtime, we will obtain similar results. These results are illustrated in Figure 4.7. Again, the network buffer occupancy increases linearly with runtime when buffering is present.

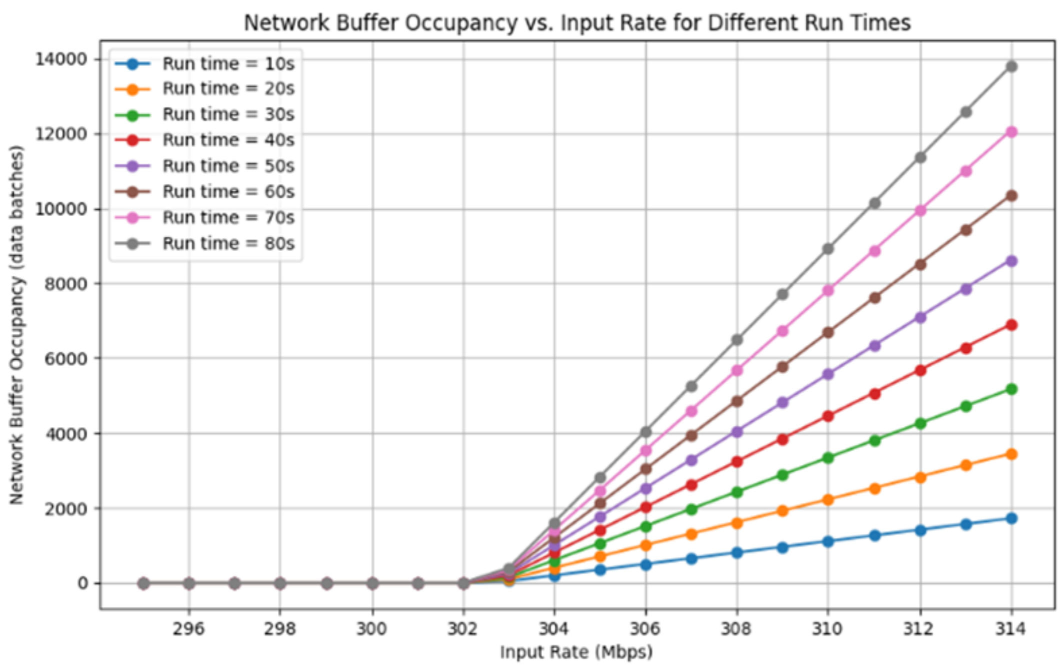


Figure 4.6: Ranging maximum occupancy with different running time (fixed output rate).

4.5 Switching Clock to 200 MHz

In designing the simulator, we focused primarily on input rate, output rate, clock speed, and runtime because the initiation interval of each processor is generally fixed. Previously, we discussed the impact of different input and outputs rates on buffer occupancy and the influence of runtime. Next, we will briefly examine the effect of clock frequency on the buffer's maximum occupancy.

Our pipeline model supports two clock frequencies: 200 MHz and 250 MHz. Therefore, we will discuss the additional case of 200 MHz in comparison to 250 MHz. As shown in Figure 4.8, the impact of clock frequency changes on buffer size is not significantly different between 200 MHz and 250 MHz (compare Figure 4.8(a) to Figure 4.3 and compare Figure 4.8(b) to Figure 4.5). The slight variation in clock speed does not substantially affect the buffer's performance, suggesting that the buffer capacity requirements remain consistent across these frequencies. Consequently, we will not conduct additional simulation testing for different runtimes under varying clock frequencies, as the current data sufficiently demonstrates that clock frequency has a minimal impact on the needed buffer size.

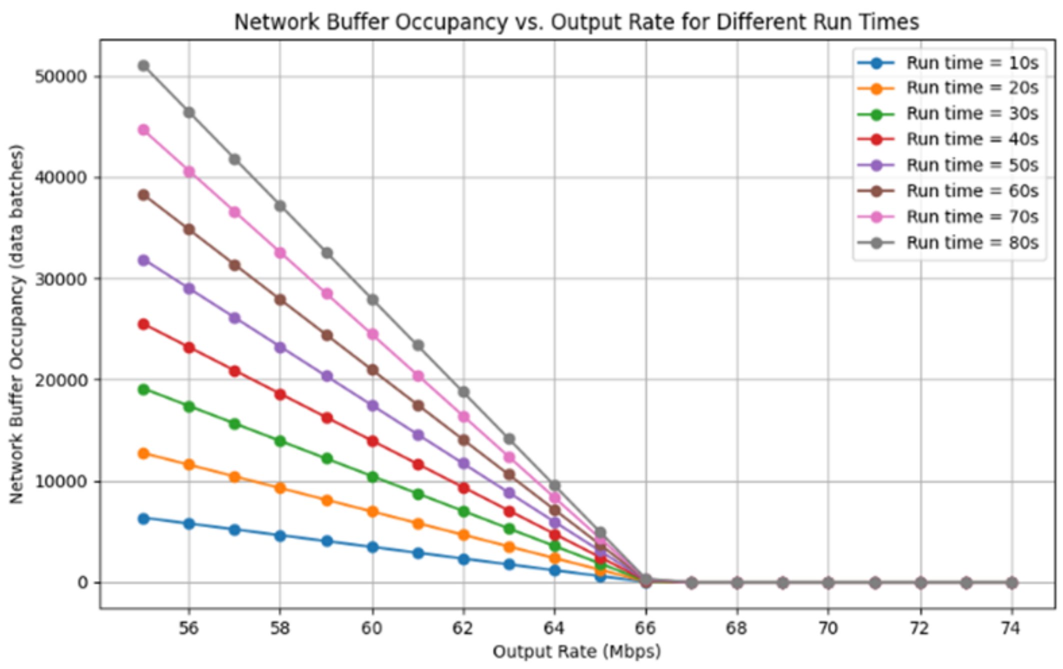
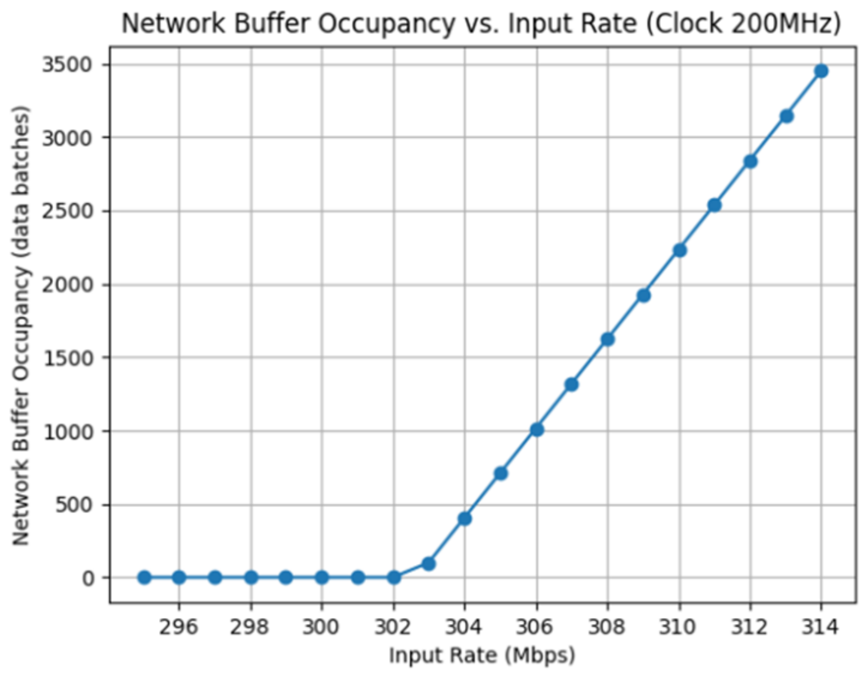


Figure 4.7: Ranging maximum occupancy with different running time (fixed input rate).

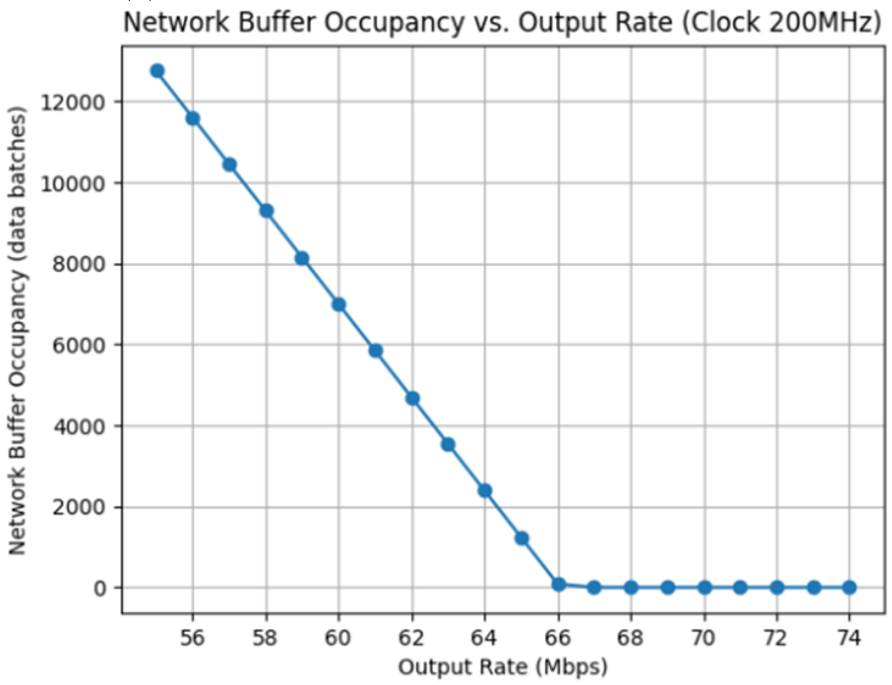
4.6 Tracking Network Buffer Occupancy with Time

To more closely observe the buffer occupancy near the breakpoint (where buffering occurs), we plotted the curve showing how occupancy changes over time. From Figure 4.3 above, we can see that for a fixed output rate, the threshold input rate is 302 Mbps. When the input rate is slightly increased to 302.673 Mbps, we observe (in Figure 4.9(a)) that during the simulation the network buffer is almost never occupied, with only a few instances where a single data packet is present. However, when the input rate is further increased to 302.68 Mbps and the simulation is run for 30 seconds (see Figure 4.9(b)), the buffer occupancy continues to rise steadily.

Building on this, we combine the curves for comparison in Figure 4.10. It is evident that all of the curves with greater than 302.67 Mbps show a relatively modest upward trend as the simulation time extends.

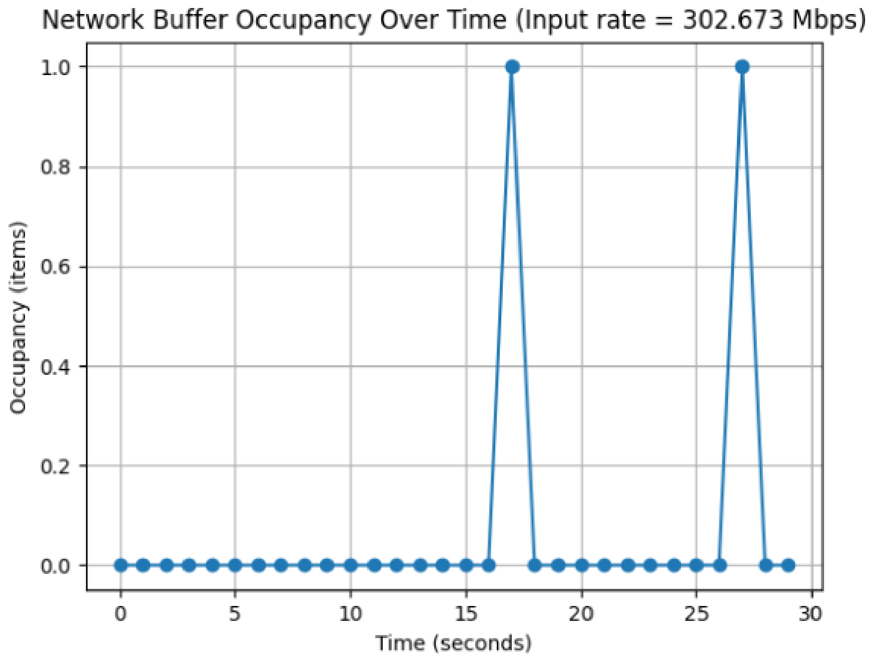


(a) Ranging input rate with a clock of 200 MHz.

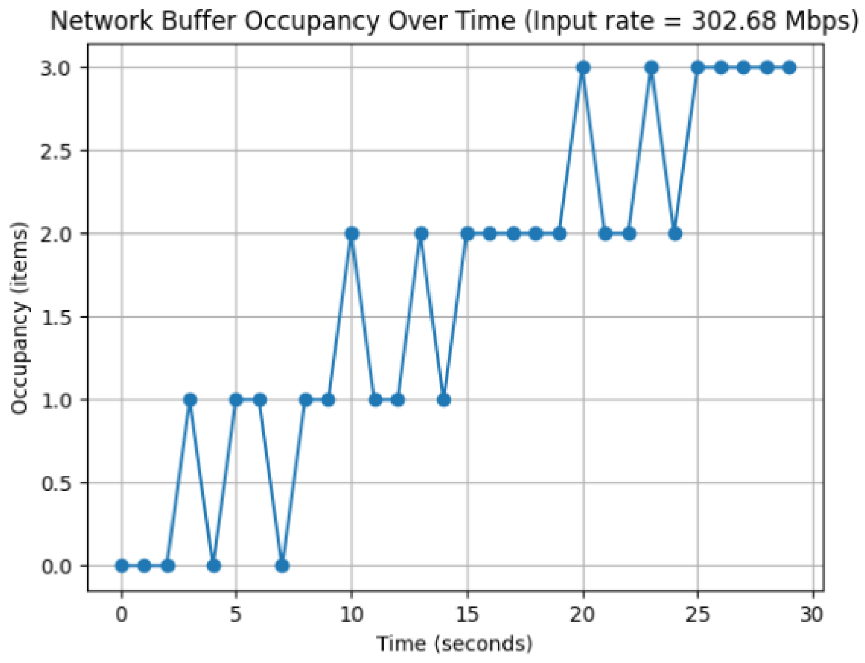


(b) Ranging output rate with a clock of 200 MHz.

Figure 4.8: Buffer occupancy with a 200 MHz clock.

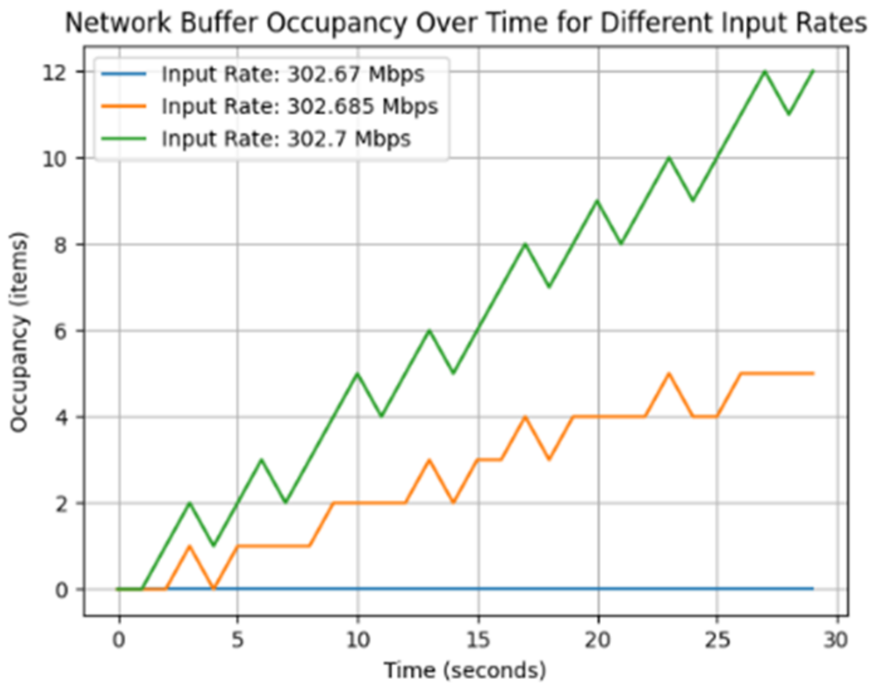


(a) Tracking occupancy with input rate 302.673 Mbps.

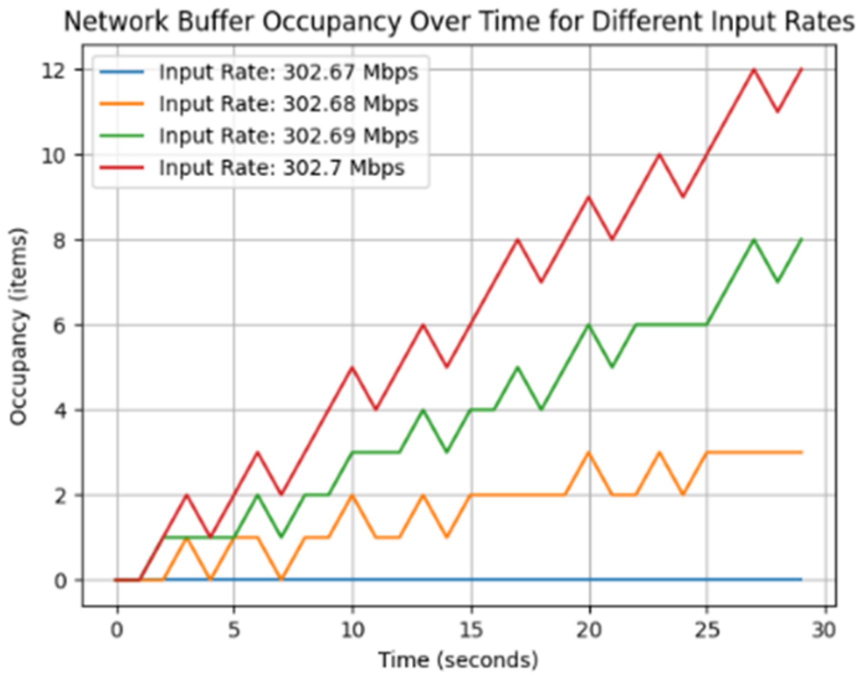


(b) Tracking occupancy with input rate 302.68 Mbps.

Figure 4.9: Network buffer occupancy increasing with time.



(a) 3 different input rates.



(b) 4 different input rates.

Figure 4.10: Network buffer occupancy with different input rates.

4.7 Comparing Network Buffer Occupancy with Time (Fixed & Poisson)

There is no substantive difference in the buffering performance between fixed interval inputs (representing the maximum rate that the ALPHA chips can provide data) and Poisson distributed inputs (for an alternative input mechanism with a higher peak rate that then reflects the physical Poisson arrival process of gamma rays) in all of the earlier presented results. However, the trace data presented in Figures 4.9 and 4.10 don't have this property. Input arrivals that are distributed via a Poisson arrival process will not look the same in their trace data as deterministic arrivals.

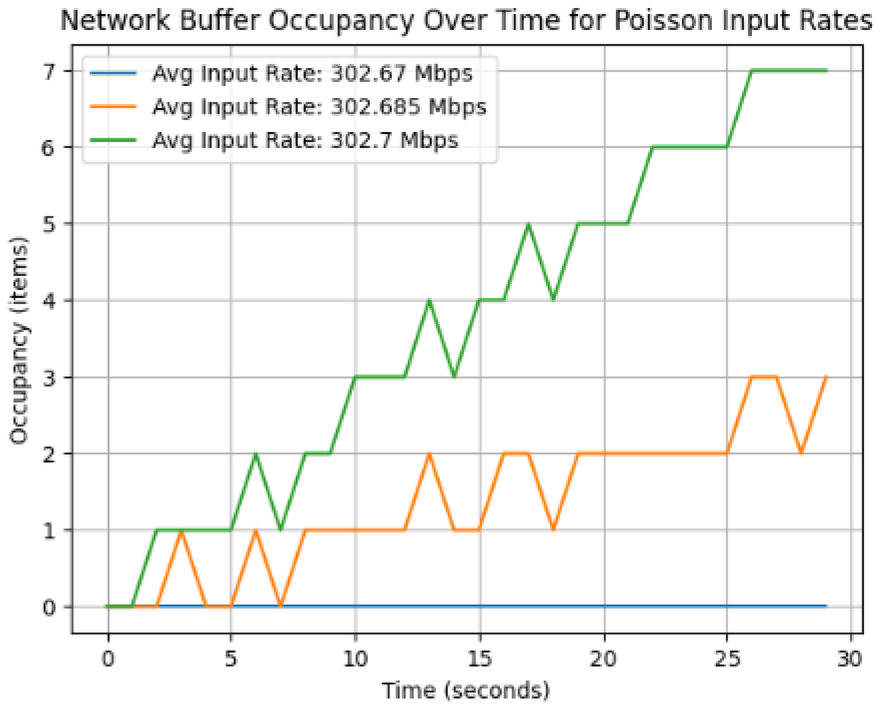
Figure 4.11 shows two plots tracing the network buffer occupancy over time with the input interarrival times being exponentially distributed (i.e., coming from a Poisson distribution). In this case, the input arrival rate is the mean arrival rate. Figure 4.11(a) can be compared to Figure 4.9 and Figure 4.11(b) can be compared to Figure 4.10.

While the detailed specifics of each plot are different, as is to be expected, the general conclusion is very much the same. Right at the same input arrival rate (whether it be deterministic or a mean value) there is a transition from a negligible buffer occupancy to a slowly building buffer occupancy in the part of the network interface buffer.

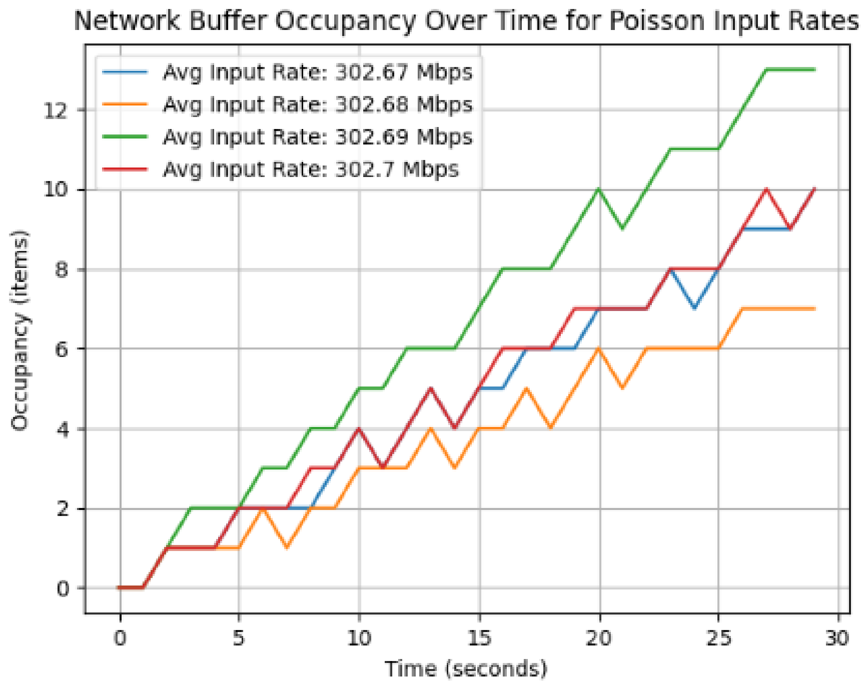
4.8 Varying the Chip-to-chip Data Rate

Up to now, the discussion has not included the chip-to-chip (C2C) channel (it is assumed to not be a limiting factor). We now introduce the C2C channel and analyze the impact of different C2C rates. The inclusion of the C2C channel has minimal effect on the buffer occupancy within the centroiding FPGA. Therefore, we focus on the occupancy of the buffer immediately preceding the C2C channel, which we will refer to as the "C2C buffer" (see Figure 4.12).

Figure 4.13 shows how the buffer occupancy of the C2C buffer varies with the C2C data rate. Above 45 Mbps, we are essentially in the same state as the models earlier in the chapter, in which the chip-to-chip link doesn't impact the performance of the pipeline. At 44 Mbps and



(a) 3 different input rates.



(b) 4 different input rates.

Figure 4.11: Network buffer occupancy with different input rates. Inputs are from Poisson distribution.

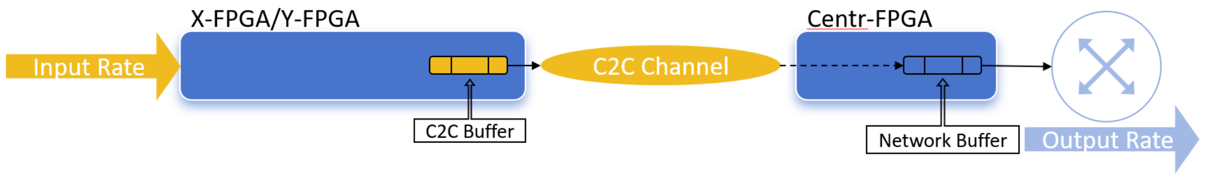


Figure 4.12: Data pipeline with changing C2C rate.

lower, we clearly get buffering in the C2C buffer. As is the case for the network interface buffer, lower C2C data rates result in increased buffer occupancy.

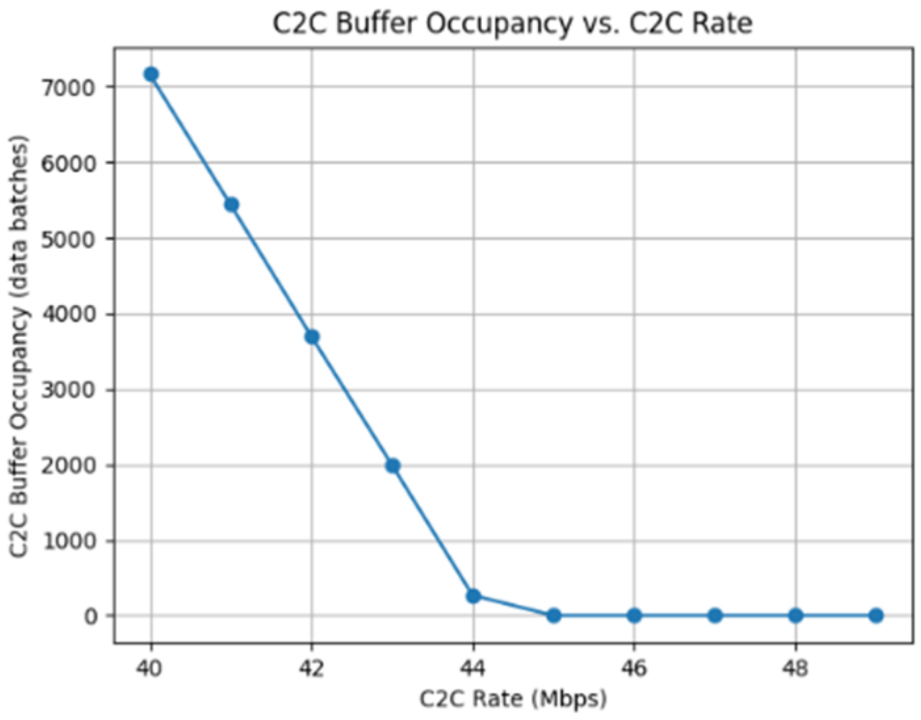


Figure 4.13: C2C buffer occupancy with varying C2C data rates.

The above limitation (when the C2C link becomes a bottleneck) is clearly also a function of the input data rate. We explore this circumstance next. Figure 4.14 plots C2C buffer occupancy against input data rate for a set of different C2C link rates. Here, we see the transition from non-limiting performance to limiting performance happen at different input data rates.

At this point, we have explored two variables—the input rate and the C2C rate. From the graph, it can be observed that with the same input rate, different C2C rates affect the

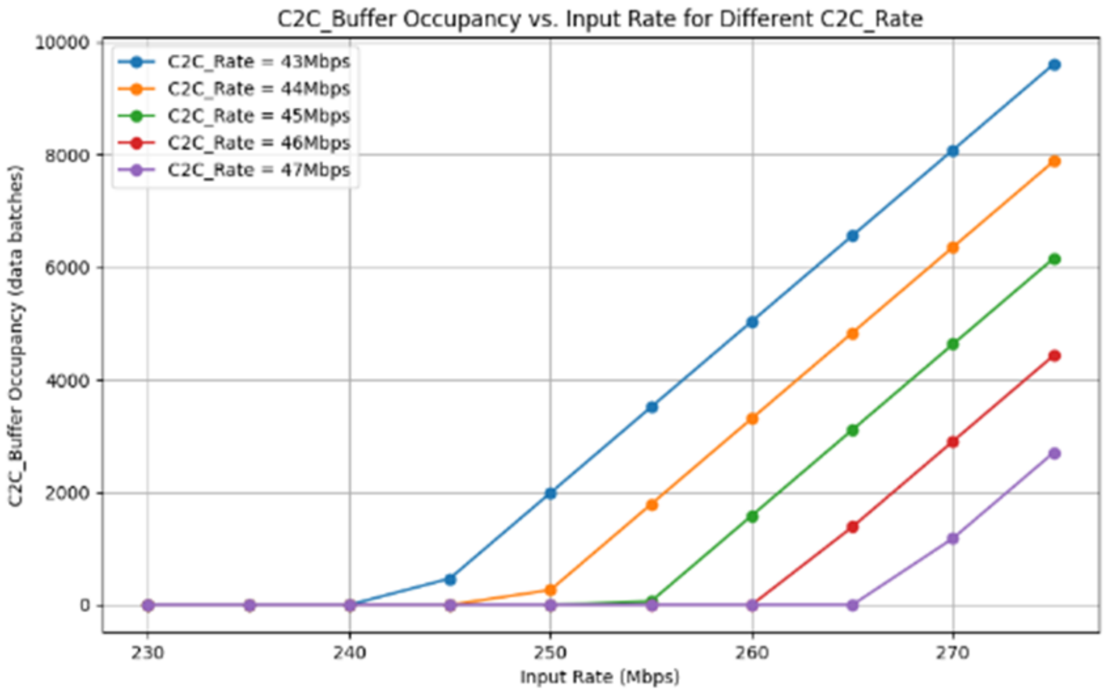
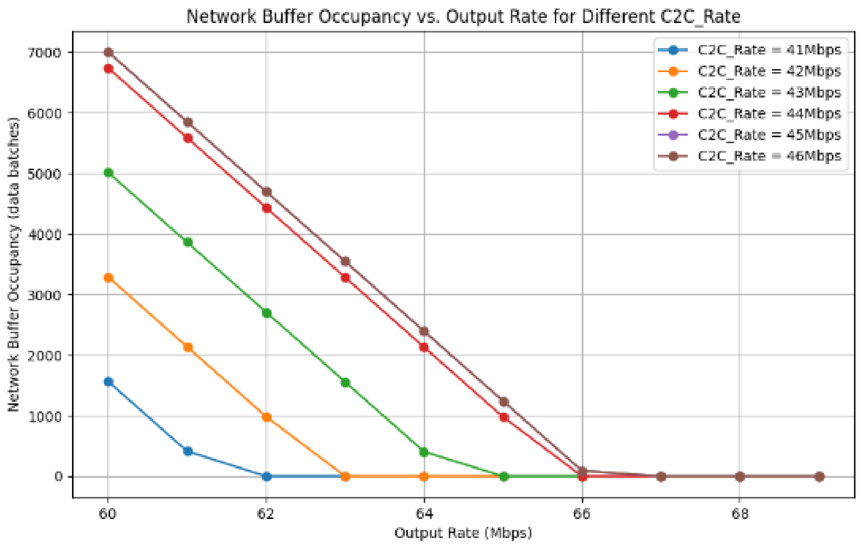


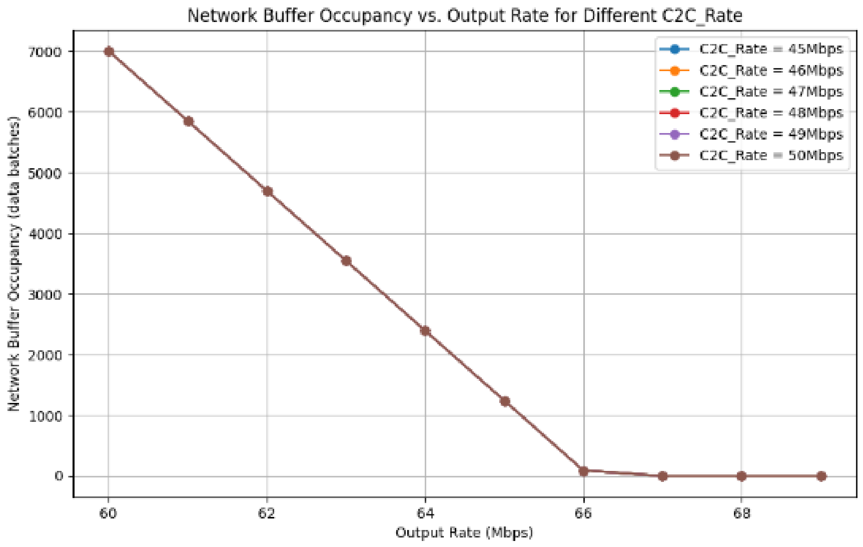
Figure 4.14: Ranging chip-to-chip rate with different input rates (output rate is 80 Mbps).

occupancy threshold of the C2C buffer. Increasing both the input rate and the C2C rate simultaneously has a relatively smaller impact on buffer occupancy.

When we switch the second variable from input rate to output rate, the input rate is set to the initial reference value of 250 Mbps. From the previous experiments, we know that when the input rate is 250 Mbps, changes in the C2C rate do not affect the occupancy of the C2C buffer. Therefore, our focus shifts back to the network buffer. As shown in Figure 4.15(a), we observe that the smaller the chip-to-chip rate, the lighter the load on the network buffer, and the lower the threshold for the output rate triggering non-empty buffers. In Figure 4.15(b), we find that no matter how much the C2C rate increases, the curve remains unchanged. This curve becomes identical to that of Figure 4.5. This is because, with the fixed initiation intervals (II) of each processing element, increasing the C2C rate no longer introduces resistance and can even be regarded as non-existent (i.e., ignored in the model), leading to this result.



(a) C2C buffer occupancy with different chip-to-chip rates.



(b) Ranging chip-to-chip rates over 45 Mbps.

Figure 4.15: Ranging chip-to-chip rates with different output rates (input rate is 250 Mbps).

4.9 Summary

The take-home message from this work is that the buffering requirements in the computational pipeline are acting effectively as if both the input arrival process and the service processes are all deterministic. This is true even when the actual arrival process is not deterministic, but rather is Poisson.

The implications of the above observation are that for each individual potential bottleneck, there is a sharp transition between "fast enough," in which there is limited to no buffering required and "too slow," in which the buffer occupancy grows without bound.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis simulates and discusses the performance of the ADAPT telescope's data pipeline. By evaluating the impact of different data transmission rates (including input rate, output rate, and chip-to-chip rate) on the pipeline's performance, we aim to determine the maximum input rate and minimum output rate that the pipeline can sustain to prevent packet loss or pipeline congestion.

During the simulations, we estimated an average input rate based on the frequency of gamma-ray generation and emission in the universe, as limited by the input rate sustainable by the front-end electronics. This rate served as a reference for adjusting the output rate (i.e., the network interface transfer rate) to assess its impact on the overall pipeline. Similarly, we fixed an initial network interface transfer rate and explored the effect of adjusting the input rate on pipeline performance. Following the same approach, when discussing the chip-to-chip rate, we kept the input and output rates constant as control variables. To prevent packet loss and congestion, we introduced an infinitely large buffer after each data processor and monitored buffer occupancy to determine the extreme sustainable rates for the pipeline.

The simulations revealed that, due to the fixed initiation interval (II) of each processing element, the first three processors had similar and significantly larger IIs compared to the latter ones. As a result, the buffers between processors remained largely unoccupied, while the buffer before the network interface consistently showed occupancy (a similar phenomenon was observed in the buffer before the chip-to-chip channel when it was included in the analysis). Therefore, our focus shifted to the buffers at these critical junctions.

Through several rounds of simulation, we determined the boundary or threshold values for different transmission rates. At the same time, we extended the simulation duration to model the buffer occupancy during prolonged data reception, aiming to find the maximum occupancy and thus confirm the minimum buffer capacity required to avoid affecting pipeline performance. Because in these cases we had exceeded the data movement capacity of the pipeline stage, regardless of how long the simulation was extended, buffer occupancy continued to grow linearly, preventing us from obtaining a bounded curve. Consequently, the system effectively operates as a pipeline with deterministic service times and a deterministic arrival process, even when the actual arrival process is Poisson.

5.2 Future Work

Building on the findings from this study, there are several avenues for future research to further enhance the understanding and performance of the ADAPT telescope's data pipeline:

1. Dynamic Buffer Sizing and Adaptive Rate Control: One of the key limitations encountered was the linear growth of buffer occupancy over time, suggesting that current fixed buffer strategies may be insufficient for long-term operations. Future work could explore dynamic buffer sizing techniques, where buffer capacity adjusts in real-time based on current data flow conditions. Additionally, implementing adaptive rate control algorithms may help optimize the input, output, and chip-to-chip transmission rates to reduce occupancy and prevent pipeline congestion.

2. Advanced Load Balancing Techniques: Since the last buffer before the network interface and chip-to-chip channel is where the occupancy issues manifest themselves, future work could explore more sophisticated load balancing strategies between processors. By distributing the data load more evenly, especially across critical junctions, it may be possible to mitigate the bottlenecks observed in the current pipeline configuration.

3. Long-Term Pipeline Behavior Modeling: While this study primarily focused on short-term simulation, extending the analysis to model long-term pipeline behavior more accurately is essential. Future work could involve advanced mathematical modeling techniques or machine learning approaches to predict buffer occupancy over extended

periods, helping identify potential long-term performance boundaries and avoiding linear occupancy growth.

4. Exploration of Alternative Processing Architectures: Given that the initiation intervals (II) of the first few processors are significantly larger than the latter ones, alternative processing architectures or task scheduling methods could be explored. Future research might focus on optimizing processor intervals to create a more balanced data flow throughout the pipeline, reducing the burden on buffers and improving overall performance.

5. Integration of Real-World Data and Feedback: This study relied on simulated data based on gamma-ray frequencies and other cosmic events. Future work could focus on integrating real-world observational data into the simulation to more accurately reflect the performance requirements of the ADAPT telescope. Additionally, incorporating feedback mechanisms from the system in real-time could further refine the pipeline's adaptability to changing data conditions.

6. Boundary and Threshold Refinement: While we determined initial boundary and threshold values for transmission rates, future research could focus on refining these thresholds through more granular simulations or by applying optimization techniques to define more precise limits for various pipeline parameters. This would enable a clearer understanding of the maximum sustainable rates without risking packet loss or system congestion.

By addressing these areas, future research can provide a more robust and scalable pipeline solution for the ADAPT telescope, ensuring it can handle varying data rates and conditions without performance degradation.

References

[1] James Buckley et al. The Advanced Particle-astrophysics Telescope (APT) Project Status. In *Proc. of 37th Int'l Cosmic Ray Conference*, volume 395, pages 655:1–655:9. Sissa Medialab, July 2021.

[2] Wenlei Chen et al. The Advanced Particle-astrophysics Telescope: Simulation of the Instrument Performance for Gamma-Ray Detection. In *Proc. of 37th Int'l Cosmic Ray Conference*, volume 395, pages 590:1–590:9. Sissa Medialab, 2021.

[3] Marion Sudvarg et al. A Fast GRB Source Localization Pipeline for the Advanced Particle-astrophysics Telescope. In *Proc. of 37th Int'l Cosmic Ray Conference*, volume 395, pages 588:1–588:9. Sissa Medialab, July 2021.

[4] Wenlei Chen, James Buckley, et al. Simulation of the instrument performance of the Antarctic Demonstrator for the Advanced Particle-astrophysics Telescope in the presence of the MeV background. In *Proc. of 38th Int'l Cosmic Ray Conference*, volume 444, pages 841:1–841:9. Sissa Medialab, July 2023.

[5] C. Aramo, E. Bissaldi, M. Bitossi, et al. A SiPM multichannel ASIC for high Resolution Cherenkov Telescopes (SMART) developed for the pSCT camera telescope. *Nucl. Instrum. Methods Phys. Res. A*, 1047:167839, 2023.

[6] K. Bechtol, S. Funk, A. Okumura, L.L. Ruckman, A. Simons, H. Tajima, J. Vandenbroucke, and G.S. Varner. TARGET: A multi-channel digitizer chip for very-high-energy gamma-ray telescopes. *Astroparticle Physics*, 36(1):156–165, 2012.

[7] Ye Htet, Marion Sudvarg, Jeremy Buhler, Roger D. Chamberlain, and James H. Buckley. Localization of gamma-ray bursts in a balloon-borne telescope. In *Proc. of Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W)*, pages 395–398. ACM, November 2023.

[8] James H. Buckley, Jeremy Buhler, and Roger D. Chamberlain. The advanced particle-astrophysics telescope (APT): Computation in space. In *Proc. of 21st International Conference on Computing Frontiers Workshops and Special Sessions*. ACM, May 2024.

[9] Marion Sudvarg, Chenfeng Zhao, Ye Htet, Meagan Konst, Thomas Lang, Nick Song, Roger D. Chamberlain, Jeremy Buhler, and James H. Buckley. HLS taking flight: Toward using high-level synthesis techniques in a space-borne instrument. In *Proc. of 21st International Conference on Computing Frontiers*. ACM, May 2024.

[10] Stefan Ritt. Design and performance of the 6 GHz waveform digitizing chip DRS4. In *IEEE Nuclear Science Symposium Conference Record*, pages 1512–1515. IEEE, 2008.

[11] M. Mishra, K. Flood, K. Lauritzen, L. Macchiarulo, I. Mostafanezhad, B. Rotter, G. Uehara, and G. Varner. Application of high density digitizer system-on-chip (HDSoC) prototype for acquiring fast silicon photomultiplier signals. In *Proc. of Nuclear Science Symposium and Medical Imaging Conference*. IEEE, 2022.

[12] Richelle Adams. Active queue management: A survey. *IEEE Communications Surveys & Tutorials*, 15(3):1425–1476, 2012.

[13] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[14] Dong Lin and Robert Morris. Dynamics of random early detection. In *Proc. of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 127–137, 1997.

[15] H. Jiang, J. Guan, J. Chen, and L. Wang. Adaptive buffer resizing for high-performance multicore systems. In *Proc. of ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2012.

[16] Chenfeng Zhao, Clayton J. Faber, Roger D. Chamberlain, and Xuan Zhang. HLPerf: Demystifying the performance of HLS-based graph neural networks with dataflow architectures. *ACM Transactions on Reconfigurable Technology and Systems*, 2024.

[17] Clayton J. Faber and Roger D. Chamberlain. Application of network calculus models to heterogeneous streaming applications. In *Proc. of IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 198–201, May 2024.

[18] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[19] S.F. Altschul, T.L. Madden, A.A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D.J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–402, 1997.

[20] R.L. Cruz. A calculus for network delay. I. Network elements in isolation. *IEEE Trans. Inf. Theory*, 37(1):114–131, 1991.

[21] R.L. Cruz. A calculus for network delay. II. Network analysis. *IEEE Trans. Inf. Theory*, 37(1):132–141, 1991.

[22] Leonard Kleinrock. *Queueing Systems, Volume I: Theory*. John Wiley, New York, NY, USA, 1975.

[23] Ahmed O. El Meligy, Mohamed S. Hassan, and Taha Landolsi. A buffer-based rate adaptation approach for video streaming over HTTP. In *Proc. of Wireless Telecommunications Symposium (WTS)*. IEEE, 2020.

[24] Jerry Banks. *Handbook of Simulation: Principles, Methodology, Advances, Applications, and Practice.* John Wiley & Sons, New York, NY, USA, 1998.

[25] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis.* Springer, New York, NY, USA, 2001.

[26] A. Law and D. Kelton. *Simulation Modelling and Analysis.* McGraw Hill, New York, NY, USA, 1991.

[27] SimPy Team. SimPy: Discrete event simulation for Python. https://simpy.readthedocs.io, 2023. Accessed Aug. 2023.