Dissertation

submitted to the

Combined Faculty of Mathematics, Engineering and Natural Sciences

of Heidelberg University, Germany

for the degree of

Doctor of Natural Sciences

Put forward by

M.Sc. Tobias Thommes

born in: Heidelberg

Oral examination: 11.12.2023

Interconnect technologies for very large spiking neural networks

Referees: Dr. habil. Johannes Schemmel

Prof. Dr. Peter Fischer

Interconnect technologies for very large spiking neural networks

In the scope of this thesis, a neural event communication architecture has been developed for use in an accelerated neuromorphic computing system and with a packet-based high performance interconnection network. Existing neuromorphic computing systems mostly use highly customised interconnection networks, directly routing single spike events to their destination. In contrast, the approach of this thesis uses a general purpose packet-based interconnection network and accumulates multiple spike events at the source node into larger network packets destined to common destinations. This is required to optimise the payload efficiency, given relatively large packet headers as compared to the size of neural spike events.
Theoretical considerations are made about the efficiency of different event aggregation strategies. Thereby, important factors are the number of occurring event network-destinations and their relative frequency, as well as the number of available accumulation buffers. Based on the concept of Markov Chains, an analytical method is developed and used to evaluate these aggregation strategies. Additionally, some of these strategies are stochastically simulated in order to verify the analytical method and evaluate them beyond its applicability. Based on the results of this analysis, an optimisation strategy is proposed for the mapping of neural populations onto interconnected neuromorphic chips, as well as the joint assignment of event network-destinations to a set of accumulation buffers.
During this thesis, such an event communication architecture has been implemented on the communication FPGAs in the BrainScaleS-2 accelerated neuromorphic computing system. Thereby, its usability can be scaled beyond single chip setups. For this, the EXTOLL network technology is used to transport and route the aggregated neural event packets with high bandwidth and low latency. At the FPGA, a network bandwidth of up to $12\,\frac{\text{Gbit}}{\text{s}}$ is usable at a maximum payload efficiency of 94 %. The latency has been measured in the scope of this thesis to a range between 1.6 µs and 2.3 µs across the network between two neuron circuits on separate chips. This latency is thereby mostly dominated by the path from the neuromorphic chip across the communication FPGA into the network and back on the receiving side. As the EXTOLL network hardware itself is clocked at a much higher frequency than the FPGAs, the latency is expected to scale in the order of only approximately 75 ns for each additional hop through the network.
For being able to globally interpret the arrival timestamps that are transmitted with every spike event, the system time counters on the FPGAs are synchronised across the network. For this, the global interrupt mechanism implemented in the EXTOLL hardware is characterised and used within this thesis. With this, a synchronisation accuracy of ± 40 ns could be measured.
At the end of this thesis, the successful emulation of a neural signal propagation model, distributed across two BrainScaleS-2 chips and FPGAs is demonstrated using the implemented event communication architecture and the described synchronisation mechanism.

Verbindungstechnologien für sehr große spikende Neuronale Netze

Im Rahmen dieser Arbeit wurde eine Kommunikationsarchitektur für neuronale Spike Events in einem beschleunigten neuromorphen Rechnersystem unter Benutzung eines paketbasierten Verbindungsnetzwerks entwickelt. Bestehende neuromorphe Computersysteme nutzen meist hoch spezialisierte Verbindungsnetzwerke, bei welchen einzelne Spike Events direkt zu ihrem Ziel geroutet werden. Dagegen verwendet der Ansatz dieser Arbeit ein allgemeines paketbasiertes Hochleistungs-Verbindungsnetzwerk und akkumuliert dazu mehrere Events zu größeren Paketen, die dann zum gemeinsamen Ziel der Events gesendet werden. Dies ist notwendig, um die Daten-Nutzlast-Effizienz, bezogen auf den relativ großen Paketheader verglichen mit einem einzelnen Event, sicherzustellen.
Es werden theoretische Überlegungen über die Effizienz verschiedener Strategien zur Akkumulation von Spike Events angestellt. Wichtige Faktoren sind dabei die Anzahl von vorkommenden Event-Netzwerkzielen und deren relative Häufigkeit, sowie die Anzahl verfügbarer Pufferspeicher zur Akkumulation. Basierend auf dem Konzept von Markov Ketten, wird eine analytische Methode zur Evaluation dieser Akkumulationsstrategien entwickelt. Zusätzlich werden einige dieser Strategien stochastisch simuliert, um die analytische Methode zu verifizieren und diese Strategien über die Gültigkeit der Methode hinaus zu untersuchen. Basierend auf den Ergebnissen dieser Analyse, wird eine Strategie zur Optimierung der Verteilung neuronaler Populationen über mehrere vernetzte Mikrochips hinweg, sowie der Zuordnung von Event-Netzwerkzielen zu einer Menge von Akkumulations-Pufferspeichern, vorgeschlagen.
Während dieser Arbeit wurde eine solche Event-Kommunikationsarchitektur auf den Kommunikations-FPGAs des beschleunigten neuromorphen Computersystems BrainScaleS-2 implementiert. Dadurch kann dessen Nutzbarkeit über die Nutzung von Einzel-Chip-Aufbauten hinaus skaliert werden. Hierfür wird die EXTOLL Netzwerk Technologie genutzt, um die aggregierten neuralen Event Pakete mit hoher Bandbreite und niedriger Latenz zu übertragen. An den FPGAs ist dadurch eine Netzwerkbandbreite von bis zu 12 $\frac{\text{Gbit}}{\text{s}}$ nutzbar bei einer maximalen Effizienz der Datennutzlast von 94 %. Die Latenz wurde im Rahmen dieser Arbeit in einem Bereich zwischen 1.6 µs und 2.3 µs über das Netzwerk zwischen zwei Neuronschaltungen auf separaten Mikrochips gemessen. Diese Latenz ist größtenteils dominiert durch den Pfad vom neuromorphen Chip über das Kommunikations-FPGA in das Netzwerk und zurück auf der Empfangsseite. Da die EXTOLL Netzwerkhardware selbst mit einer viel höheren Taktrate als die FPGAs betrieben wird, ist zu erwarten, dass die Netzwerklatenz in der Größenordnung von lediglich etwa 75 ns pro zusätzlichem Netzwerkschritt skaliert.
Um die Ankunftszeitstempel, welche mit jedem Spike Event versendet werden, global interpretieren zu können, werden die Systemzeitzähler der FPGAs über das Netzwerk synchronisiert. Dazu wird im Rahmen dieser Arbeit der globale Interruptmechanismus des EXTOLL Netzwerks charakterisiert und verwendet. Damit konnte eine Synchronisationsgenauigkeit von ± 40 ns gemessen werden.
Am Ende dieser Arbeit wird die erfolgreiche Emulation eines Modells zur neuronalen Signalweiterleitung, verteilt über zwei BrainScaleS-2 Chips und FPGAs hinweg, unter Verwendung der entwickelten und implementierten Event Kommunikationsarchitektur und des beschriebenen Synchronisationsmechanismus, demonstriert.

Table of Contents

Table of Contents vii

I Introduction 1

1 Motivation and Overview 3

2 Background 13

2.1 Neural Networks in Biology . . . 13
2.2 Neuromorphic Computing . . . 16
2.3 High Performance Interconnection Networks . . . 20

3 The BrainScaleS-2 System 33

3.1 The HICANN-X ASIC . . . 34
3.2 The Communication FPGA . . . 37
3.3 The Software Stack and Experiment Flow . . . 43

4 The EXTOLL Network Technology 47

4.1 The Network Partition . . . 48
4.2 The NIC partition . . . 52
4.3 The Host Interface . . . 54
4.4 The Software Stack . . . 55

II Event Communication 57

5 Event Communication Principles and Systems 59

5.1 Event Communication in General . . . 59
5.2 Event Communication for SNNs . . . 60
5.3 Quality of Service Requirements for Spike Communication . . . 61
5.4 Methods for obtaining Spike Communication Quality of Service . . . 64
5.5 Existing Spike Communication Architectures . . . 65
5.6 Event Communication in a Packet-Based Network . . . 67

6 Formal Analysis of Event Aggregation 75

6.1 Accumulation Buckets . . . 76
6.2 Mathematical Analysis . . . 81
6.3 Results of the Mathematical Analysis . . . 91
6.4 Simulation Analysis . . . 102

III Implementation and Experiments 107

7 The Implemented Event Communication 109
7.1 The Event Switch . . . 110
7.2 Systime Synchronisation . . . 116
7.3 Event Transmission . . . 118
7.4 The NHTL Transaction Layer . . . 122
7.5 Event Reception . . . 124
7.6 Configuration and Status Interfaces . . . 129
7.7 Clock Domain Signal Synchronisation . . . 137
7.8 Design Parametrisation . . . 139

8 Commissioning 141
8.1 Simulation and Verification . . . 141
8.2 Physical FPGA implementation . . . 148
8.3 Network Operation Tools . . . 156
8.4 Software Integration . . . 159
8.5 Inter-Chip Latency Measurement . . . 166
8.6 Systime and Experiment Synchronisation . . . 169
8.7 The Synfire Chain Experiment . . . 181

9 Conclusion 189
9.1 Summary . . . 189
9.2 Outlook and Discussion . . . 194

IV Appendix 199

A Mathematics derivations 201
A.1 Derivation of the Dennard Scaling Law . . . 201
A.2 Poisson Distribution Statistics . . . 202
A.3 Proving total probability in the Markov Transition Matrix . . . 203

B Implementation Details 205
B.1 Used Signal Interfaces . . . 205
B.2 Used Network Packet Types . . . 208

C Dynamic Bucket Concept 213
C.1 Overview . . . 213
C.2 Arbitration Request Pipeline . . . 215
C.3 Bucket Finite-State Machine . . . 216

D Acronyms 219

Publications 226

References 227

Acknowledgements - Danksagungen 242

Part I

Introduction

1 Motivation and Overview

Ever since the first computing machines were built during the time of the second world war by Konrad Zuse (Z3, 1941) and Alan Turing (The Letchworth-Enigma (1940) (Sale 2004), The Turing Bombe (Davies 1999)), computers have become increasingly powerful. Starting with mechanical constructions as the Letchworth-Enigma or Zuse's Z1 (Wikipedia 2023d), technology rapidly transitioned to electromagnetic relays with the Z2 (Wikipedia 2023e). The first vacuum-tube based computer was the Electronic Numerical Integrator and Computer (ENIAC) (1946), developed at the University of Pennsylvania (Eckert et al. 1964). In 1954 the TRAnsistor Digital Computer (TRADIC) (Irvine 2001) was the first computer to be based on discrete transistor technology.
With the invention of the first integrated circuits by Jack Kilby based on Germanium (Kilby 1964) and Robert Noyce based on Silicon (Noyce 1961) in 1959, the development in computer technology rapidly advanced to increasingly high transistor integration densities, while rapidly shrinking the technological feature size of transistors. While in 1971 the smallest feature size on an integrated circuit was around 10 µm (Wikipedia 2023a), current CMOS process nodes of different manufacturing companies such as TSMC or Samsung offer transistors feature sizes around 5 nm [2] and metal pitch sizes of around 24 nm (IEEE 2021; Wikipedia 2023b).

The Dennard scaling law (Dennard et al. 1974) connects the scaling of the physical size d of a MOSFET transistor to its performance characteristics like its power consumption P and switching frequency f. The derivation starts at the fact that electric power consumption for loading a capacitance is proportional to the value of that capacitance, as well as to the frequency and squared voltage with which it is repeatedly driven. Finally, the Dennard law arrives at the statement that power consumption is proportional to the capacitors area (for the derivation cf. Appendix A.1). Consequently the power-density will stay constant while increasing the integration density and operation frequency, i.e. the number of transistors per area. This is closely related to Moore's law, roughly postulating an ever increasing integration density while technology advances over time (Moore 2006). Figure 1.1 shows actual numbers of transistors on microchips from the years between 1970 and 2020, supporting the hypothesis of Moore's law.
However, the Dennard scaling does not take into account the transistors leakage currents and quantum tunnelling effects that become increasingly important when scaling towards the atomic level of a few nanometres length. Consequently, modern technology improvements cannot only rely on shrinking feature size to increase the frequency, but have to make further advances regarding the shape of transistors. Examples for this trend are technologies like FinFET (Jurczak et al. 2009), where multiple gates are placed on top, beside or surrounding the channel between source and drain,

[2] the width of a FinFET fin

Moore’s Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

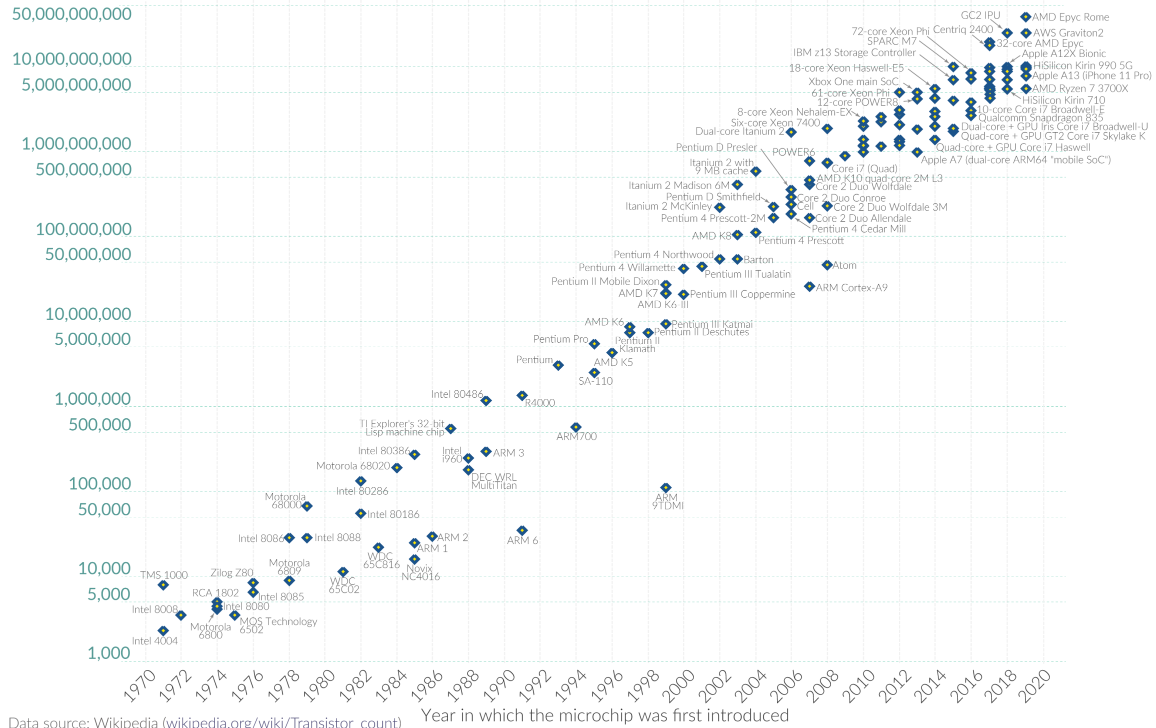


Figure 1.1: Data graph, showing the number of transistors on microchips in the years between 1970 and 2020. Figure taken from (Roser et al. 2020).

that is formed like *fins*. The latter, where the gate surrounds the transistor channel are called Gate All Around Field Effect Transistors (GAAFETs) (Y.-C. Huang et al. 2017) and pose a further technological improvement.

Another important strategy of further scaling the performance of integrated circuits and processing power is massive parallelisation instead of further pushing forward the performance of single processing units. This approach makes use of more and more processing units, cooperatively solving computing tasks.

By dividing a task into multiple sub-tasks executed in parallel, one could think that while increasing the number n of execution units working in parallel, the computation time of the overall task would scale down linearly. However, Amdahl’s Law (Amdahl 1967) states that the speed-up η, theoretically reachable by parallelisation is fundamentally limited by the sequential portion t_S and the parallel portion t_P of the tasks computational runtime $T = t_s + t_p$. Thereby, the sequential portion contains all the operations of the task that cannot be distributed amongst the processing units, as they immediately depend on the results of their preceding operations:

$$\eta = \frac{1}{s + \frac{p}{n}} \leq \frac{1}{s} \tag{1.1}$$

where $s = \frac{t_s}{T}$ is the fraction of sequential runtime and $p = \frac{t_p}{T}$ is the fraction of parallelisable runtime. In the limit of infinitely many parallel threads $n \to \infty$, Equation (1.1) converges to a maximum speed-up of $\frac{1}{s}$, limited by the sequential part of the task. This limitation can be avoided by also scaling the problem size of the task with the number of parallel processors. For this case Gustafson's law (Gustafson 1988) states, according to Amdahl's law that with double the number of processors one can compute double the amount of data in the same amount of time.

However, this analysis is still quite idealised, as Amdahl argued that with a growing number of parallel processors, there is also a rising amount of communication and synchronisation needed in order to cooperatively complete the task in parallel. This is taken into account by adding to the equation the portion $t_c(n) = \mathcal{O}(n)$ of execution time that is spent with communication ($T = t_s + t_p + t_c$).

$$\eta = \frac{1}{s + c(n) + \frac{p}{n}} \leq \frac{1}{s + c(n)} \tag{1.2}$$

Now, in the limit of infinitely many parallel threads, the speed-up no longer converges to $\frac{1}{s}$. Instead, the speed-up now exhibits an optimum point, depending on the communicational fraction $c(n) = \frac{t_c(n)}{T}$.

These considerations emphasise the need for efficient high-performance interconnection networks for the massively parallel supercomputers. Important performance indicators for interconnection networks include bandwidth, latency, message rate and availability. While *bandwidth* describes the amount of data that can be transmitted per unit of time, *latency* describes the amount of time it takes for messages to pass the network. The *message rate* on the other hand measures the number of distinct messages that can be injected in a network per unit of time while the availability characterises whether messages being transmitted along a specific route may block other messages that need to use (parts of) the same route to reach another destination. Also for data transfer across the global internet, these performance indicators are of high importance, although their particular importance largely depends on the concrete application. A more detailed introduction on interconnection networks and their design space will be presented in Section 2.3.

The Top500 organisation[1] regularly lists the worlds largest and most powerful supercomputers since 1993. The current list of June 2023 (*Top 500 List* 2023) awards the pole position to the *Frontier* system, located at the *Oak Ridge national Laboratory* in the United States. This supercomputer makes use of 8,699,904 processing cores, thereby offering a peak computing power of 1.679 EFlop/s while drawing a total electric power of 22.7 MW. Remarkably, it is thereby the first exascale computer ever built. As an interconnect it uses the *Hewlett Packard Enterprise (HPE) Slingshot* network, which has been developed by Cray (De Sensi et al. 2020). This high-performance interconnection network provides switching hardware with 64 ports, each offering a bandwidth of 200 $\frac{\text{Gbit}}{\text{s}}$ and message rates of up to 600 $\frac{\text{MMsg}}{\text{s}}$ in both directions. Adaptive routing and an active congestion management provide high availability rates. The switching latency is reported to be in a range between 300 ns and 400 ns. (De Sensi et al. 2020; Hewlett Packard Enterprise 2023)

The second rank on the current Top500 list goes to the *Supercomputer Fugaku* at the *Fujitsu RIKEN*

[1] www.top500.org

Center for Computational Science in Japan. This machine employs 7,630,848 processing cores, offering a peak computing power of 537.2 PFlop/s by drawing even more electric power of 29.9 MW. As network it uses the *Tofu Interconnect D* (Ajima et al. 2018). It offers peak injection rates of up to 300 $\frac{\text{Gbit}}{\text{s}}$ per link and a sustained throughput bandwidth of 50 $\frac{\text{Gbit}}{\text{s}}$. The switching latency is reported between 490 ns and 540 ns. (Ajima et al. 2018)

As these two examples show, the continued increase in computing power comes at the cost of enormous electric power consumption, although the power efficiency has made a huge step forward between these two examples, now finally reaching the exascale of computing power (Shalf et al. 2011).

While digital computers are mainly good at processing huge amounts of numbers and data, the human brain performs extremely complex tasks like decision making and pattern recognition. It can easily succeed in driving vehicles through crowded places and most successfully avoid harmful collisions (if not hampered by strong tiredness or drugs). Not to mention creative tasks like composing and interpreting of music, writing literature or creating beautiful paintings, and especially doing science about the world it is living in.

While reproducing these tasks using digital computing hardware uses up huge amounts of computing power while drawing correspondingly high amounts of electric power, all these tasks are performed by the human brain while only drawing an approximate continuous power of about 20 W (Leonard et al. 1994; Markram 2012). The wish to understand how this miraculously efficient and powerful organ works, and before its role was known, the question where the human behaviour, intelligence and feelings arise from, has at all times driven the field of neuroscience. Since about roughly more than a century, scientists begin to understand with huge advances, the principles of how the nervous system works. While ancient reports indicate that a rudimentary interest and understanding in the role of the brain as the origin of human behaviour and thought has already existed in ancient egypt and greece (Wikipedia 2023c), Luigi Galvani was the first modern scientist to recognise the electric nature of nerves around 1790, describing them as "animal electricity" (Piccolino 1998). Notably, his experiments with different metals contacting dissected frogs' legs led to the invention of chemical batteries by Alessandro Volta around 1800. Continuing the path of physiological studies, Charles Bell first described the difference between motor- and sensor neurons in 1811 (Grzybowski et al. 2007).

The first real breakthrough in neuroscience was achieved by Camillo Golgi and Santiago Ramón y Cajal in discovering the cellular structure of the nervous system. Cajal had used and improved colouring methods developed by Golgi in order to investigate nerve fibres from the grey brain matter and the fine structure of the retina (Hanser et al. 2000). Although Golgi and Cajal disagreed about the detailed interpretation of their findings (Jones 1999), they were both awarded the 1906 Nobel prize in Physiology and Medicine "in recognition of their work on the structure of the nervous system" (Nobel Prize Outreach AB 2023).

Another important breakthrough was made by Alan Lloyd Hodgkin and Andrew Fielding Huxley who, together with John Carew Eccles, received the 1963 Nobel prize "for their discoveries concerning the ionic mechanisms involved in excitation and inhibition in the peripheral and central portions

of the nerve cell membrane" (Nobel Prize Outreach AB 2023). The Hodgkin-Huxley neuron model (Hodgkin et al. 1952) which is named after them, quantitively describes the mechanism of how neurons produce and transfer *action potentials*, also referred to as *spike events*. A rough summary of the basic bio-physiological function of neurons and synapses will be presented in Section 2.1.

In recent years (since 2013), the neuroscientific community has collaborated in two large projects: on the west side of the Atlantic the Brain Research Through Advancing Innovative Neurotechnologies® Initiative (BRAIN Initiative), funded by the United States government, and on the east side the Human Brain Project (HBP), funded by the European Union.
In the last ten years up to its end in September 2023, the HBP has succeeded in many ways to provide a "research infrastructure that [allows] scientific and industrial researchers to advance our knowledge in fields of neuroscience, computing, and brain-related medicine"[1].
In the field of neuroscience, for example the *Brain Atlas* provides a computational framework for creating a 3-dimensional map of the cellular structure of not only the human brain, but also different animal species like e.g. mice or monkeys (Amunts et al. 2020; Silvestri et al. 2021; Stacho et al. 2020)[2].
For brain-related medicine, the HBP e.g. provides a framework for creating very detailed duplicate models of patients' brains. These help e.g. in surgery of epileptic patients to identify the exact location of those regions causing the epileptic seizures. Based on these models, surgeons can precisely plan the surgical intervention for removing this and only this affected portion of the brain (Wang et al. 2023). Besides this, neural implants have been developed for blind or paraplegic patients which are currently in clinical trial in the form of case studies (Fernández et al. 2021; Wagner et al. 2018).
In the field of robotics, the Neuro-Robotics Platform (NRP) has been developed and provided by the HBP[3]. With the help of this platform, e.g. robots have learned improved models for location remembering and navigation (Pearson et al. 2021).
Last but not least, in the area of computing, the HBP funded the development, build-up and operation of two large-scale neuromorphic computing systems, SpiNNaker and BrainScaleS[4].

Generally, neuromorphic computing aims to build computing systems that are based on the known aspects of how biological nervous systems work. A review on the history and landscape of neuromorphic computing with a focus on large-scale systems is given in (S. Furber 2016).
In principle, there are two kinds of neuromorphic computing systems. On the one hand there are those which numerically solve the differential equations of neurons and synapses to digitally simulate a neural network. On the other hand, analogue neuromorphic systems don't explicitly solve these dynamic equations, but rather implement electronic circuits that adhere to the same dynamics and thereby emulate the behaviour of neural networks. One great advantage of analogue systems is

[1] The principal self-description of the HBP on its website `https://www.humanbrainproject.eu/`
[2] These atlases are publicly available at `https://julich-brain-atlas.de/`
[3] The NRP is publicly available at `https://neurorobotics.net/index.html`
[4] Both systems can be accessed via the EBRAINS platform at `https://www.ebrains.eu/modelling-simulation-and-computing/computing/neuromorphic-computing/`

that they can be designed and dimensioned in a way that the neural dynamics run at high speedup factors, as compared to biological timescales. In contrast, depending on the level of simulated detail and available processing power, a numerical neuromorphic simulation can run in an order of 100 times slower than biology. A short introduction on the basic principles of neuromorphic computing will be presented here in Section 2.2.

As large scale neuromorphic computing systems are a special kind of supercomputers, however with a very specific workload, they also have the need for efficient interconnection networks to communicate neuromorphic event data. As biological neurons exchange information solely based on the timing, frequency and spatial distribution of so-called *action potentials* or *spike-events*, the simulation or emulation of such neural networks poses unique requirements to an interconnection network carrying the communication messages in a neuromorphic computing system. These requirements will be explained and related to existing neuromorphic computing systems like SpiNNaker and BSS-1, as well as its predecessor *Spikey* in Chapter 5.

The SpiNNaker system (S. B. Furber, Galluppi, et al. 2014), as it is specifically designed for this workload and trades numerical precision for energy efficiency and scalability, is able to digitally simulate 460 million neurons in biological realtime (S. Furber 2016). For this purpose it uses one million ARM cores and a special purpose interconnection network.

The BrainScaleS system on the other hand, as developed at the Electronic Visions Group in Heidelberg, is a mixed signal accelerated neuromorphic computing system. As such it emulates the dynamics of neurons and synapses as abstracted from biological into physical models using analogue electronic circuits. Spike events generated from these full custom integrated circuits are digitised and communicated between units of the system. Because of the fast intrinsic timescales of the electronic circuits, the analogue neuron and synapse dynamics are accelerated by factors between 10^3 (with the current BSS-2 ASIC) and 10^5 (with its historical predecessor *Spikey*) as compared to biological timescales.

As part of the HBP, the first generation BSS-1 (J. Schemmel et al. 2008; Johannes Schemmel, Brüderle, et al. 2010; H. Schmidt et al. 2023) was developed as a large scale neuromorphic computing platform. The BSS-1 System is based on the High Input Count Analog Neural Network (HICANN) chip, featuring 512 analogue neuron circuits and 224 synapses each. The HICANN was designed for neural network emulations with a speed-up factor of up to 10^4 times faster than biology. This speed-up enables investigation of the dynamics of learning processes in the order of seconds which would naturally take multiple hours.

An important feature of the BSS-1 system is its large scale design using Wafer Scale Integration (WSI). As most modern microchips, HICANNs are produced on silicon discs, called wafers. Usually individual dies are produced by cutting the wafer after production. Due to technical reasons, microchips can only be produced in area units of a certain maximum size. This maximum production area is called a reticle and is iterated over the whole area of the silicon wafers. With BSS-1 one reticle contains 8 HICANN ASICs, directly edge-connected to each other. The otherwise isolated reticles are interconnected by additional metal layers, added in a custom post-processing step at the Fraunhofer IZM in Berlin (Zoschke et al. 2017). With this technique, a BSS-1 Wafer Module

combines 48 reticles and thereby features 196,608 neuron circuits with 44,040,192 synapses. In the work of (Thanasoulis 2019), an interconnection network has been proposed in order to interconnect several of these Wafer Modules.

These numbers seem quite high, but are actually still rather small when comparing them to the approximate order of 10^{11} neurons and 10^{14} synapses in the human brain, illustrating its high complexity. In order to match these numbers of a human brain, more than 50,000 of these wafer modules would be required only for the neurons. However, to also match the vast connectivity of on average 1000 synapses per neuron, one would need approximately 200,000 of these units. To further illustrate these numbers, one can also compare to the number of approximately 150,000 neurons in a Drosophila fruit fly.

In parallel to the development of the wafer-scale system, the HICANN ASIC was further improved and advanced to the next generation, now called BrainScaleS-2 (Pehle et al. 2022). In contrast to BSS-1, which is built using a UMC 180 nm technology, the current ASIC, called High Input Count Analog Neural Network with HAGEN Extensions (HICANN-X) is manufactured in a 65 nm process technology by TSMC. Additionally, the HICANN-X ASIC features a lot of new circuits and improvements of existing ones. The probably most important advancement compared to BSS-1 is the introduction of two custom embedded SIMD microprocessors that can be used to change the extensive parametrisation of the analogue emulation circuits during their accelerated operation. Thereby, various plasticity rules can be directly programmed to the chip to investigate learning dynamics in the accelerated neuromorphic system. The acceleration, as compared to BSS-1 was reduced to a factor of 1000 in order to relax the communication bandwidth requirements.

At the current stage of development, BSS-2 has not yet developed to a full large-scale system, but offers flexible access to approximately two dozens of single chip setups. These consist of two neuromorphic ASICs, each connected to an FPGA which offers high-speed communication, bridging the gap to the user-space software on a conventional host-computer (cf. Chapter 3). The goal of this thesis is to develop, implement and test a communication and synchronisation mechanism that can be used to interconnect the BSS-2 system in a scalable manner.

Each of the existing neuromorphic computing systems reviewed in (S. Furber 2016), including SpiNNaker and BSS-1, realised a custom network implementation, optimised for the unique constraints and requirements of neural event communication. However, in the scope of this thesis another approach is taken, making use of an existing, packet-based *general-purpose* interconnection network. The main characteristic of such packet-based networks is that they always require some significant amount of management information to be transmitted in the header of each packet in order to route it towards its destination. Usually, this is significantly more than is required to route single spike events in a special purpose neuromorphic interconnect. Therefore, it is desirable to collect multiple spike events and route them in a shared packet towards their common destination in the network. For this purpose, multiple output packet buffers, referred to as *buckets* will be used.

An interconnection network, suitable for the task of neuromorphic event communication in an accelerated system should offer a good combination of high transmission bandwidth, as well as high injection rates of small packets. Additionally, the transmission latency should be as low as possible

in order to cope with the accelerated neuromorphic computation. Chapter 5 will elaborate further on these requirements.

For this task, the EXTOLL network is found suitable. The EXTOLL interconnection network has been developed by the EXTOLL company, based in Mannheim[1] which is a spin-off from the Computer Architecture Group (CAG) at the University of Heidelberg[2]. The design goals were to provide high bandwidth and message rates with low latencies for interconnecting high performance computing clusters. EXTOLL implements a direct, switch-less network, capable of connecting up to 2^{16} nodes in a 3D-Torus or any other topology with a node-degree up to 6. Multicast communication is supported by hardware with up to 64 multicast groups. A built-in barrier and interrupt mechanism supports global interrupts across the network with very low skew of only a few clock cycles. This interrupt mechanism will play an important role in this thesis in synchronising BSS-2 systems, as described in Section 7.2. The accuracy of the interrupt operation's synchrony will be evaluated in Section 8.6.3. Chapter 4 will elaborate on the distinct features of the EXTOLL network ASIC.

Overall, this thesis is organised in four parts. Starting with an introduction in Part I, an overview on the biological and technological background regarding biology, neuromorphic computing and the characterisation and design space of high-performance interconnection networks will be given in Chapter 2. The characteristics and features of the BrainScaleS-2 neuromorphic computing system, as well as the EXTOLL network will be given in Chapter 3 and Chapter 4 respectively.

In Part II, Chapter 5 first elaborates on the general principles of event communication and the requirements and strategies that follow for the implementation of event communication in a packet-based network in the context of existing literature. Chapter 6 will present a detailed formal analysis of event aggregation into packets under the constraints imposed on this task by the principles of neuromorphic event communication. The main question addressed here, is how many events can be expected to be accumulated in a packet until constraints require sending it. This is especially non-trivial in the case when there are more event destinations in the neural network, than output buffers for accumulating them. This analysis will be based on a mathematical representation of the problem using a Markov Chain model, accompanied by a stochastic simulation of the accumulation process.

In Part III, Chapter 7 deals with the implemented event communication architecture for BSS-2 and how multiple BSS-2 systems can be synchronised for this purpose, using the EXTOLL network. After having described the implementation details, Chapter 8 describes the tools and methods used for commissioning of the event communication architecture. This will include measurements of the transmission latency between two BSS-2 neuromorphic ASICs (Section 8.6.1) as well as the accuracy of synchronisation using the EXTOLL global interrupt mechanism (Section 8.6.3). Finally, the successful implementation of a Synfire Chain model on two BSS-2 neuromorphic ASICs, seamlessly working together and exchanging spike events, shows the successful implementation and integration of the described communication architecture in the existing BSS-2 system infrastructure (Section 8.7). Chapter 9 will summarise and conclude the results of this thesis and elaborate on future extensions and improvements of the described event communication architecture.

[1] www.extoll.de/

[2] www.ziti.uni-heidelberg.de/ziti/en/institute/research/computer-architecture-group

The Appendix in Part IV contains some mathematical derivations (Appendix A), as well as the documentation of some important implementation details (Appendix B). Appendix C finally describes a concept for the implementation of a dynamic event aggregation architecture, which was developed in the early time of this work, but has not been realised due to its complexity and in favour of the more simple and static approach described in Chapter 7.

2 Background

2.1 Neural Networks in Biology

In Biology, almost every complex organism is controlled by some more or less sophisticated nervous system. The main building blocks of a nervous system are excitable cells, called neurons. This Section will give a brief summary of the biophysiological processes driving the functionality of neurons and their interaction. A detailed and easily understandable review on this topic has already been given in Chapter 2 of (Petrovici 2016). The reader shall be referred to this excellent work for details and mathematical models derived from biological observations.

2.1.1 Neurons

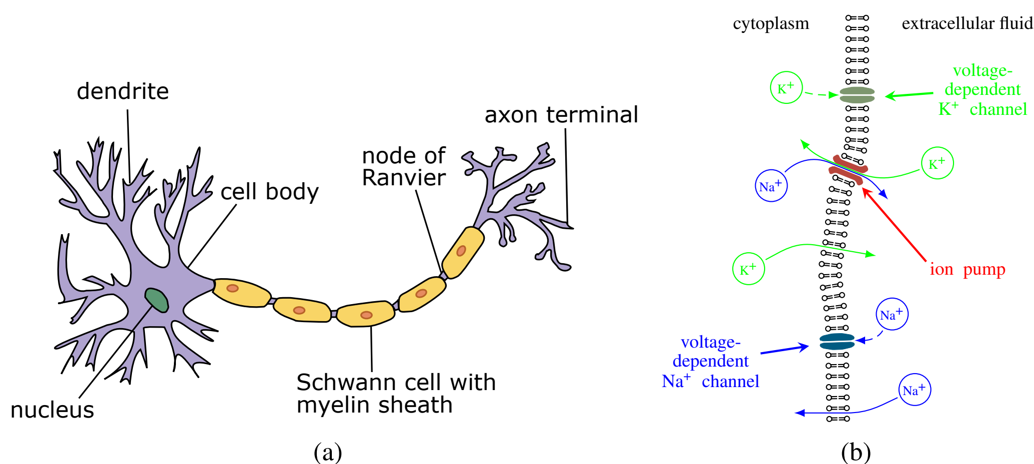


Figure 2.1: **(a)** Schematic drawing of the generic neuron structure; taken from (WikimediaUser 2019). A neuron extends from the *cell body* in several input- and output branches called *dendrites* and *axon terminals* respectively. Before branching out to multiple targets, an axon may bridge a long distance. The conductivity of these *wires* can be improved by a so-called *myelin sheath* which is produced by *Schwann cells* surrounding the axon. Spots on the axon, not affected by the *myelination* are called *nodes of Ranvier*.
(b) Schematic drawing of a neurons' cell membrane at rest; taken from Figure 2.1 (B) (Billaudelle 2022). Charged ions in liquid solution are separated between the *cytoplasm* and the *extracellular fluid*. Ions can move in and out the cell by diffusion through passive- and *voltage dependent channels* as well as active *ion pumps*, implemented by specific protein molecules in the membrane.

The overall structure of a neuron cell and its membrane is depicted in Figure 2.1. Neurons process information in the form of electrical charges, voltages and currents in the form of charged ion concentrations. The most abundant but not exclusive ion types in neurons are single-charged sodium

(Na^+) and potassium (K^+), as well as calcium (Ca^{++}) at the synapses (cf. Section 2.1.2). The respective ion concentrations inside and outside the cell are separated by the cell membrane which exhibits special protein molecules that allow for specific ion types to cross the line by either passive diffusion or being actively pumped in a specific direction. Passive diffusion channels are partly also gated by voltage dependent mechanisms in the channel proteins. Under equilibrium conditions, depending on the neuron type, the resting potential across the membrane lies between -100 mV and -50 mV, typically at -70 mV for human cells (Gekle et al. 2015).

Structurally, neurons stretch out in a *dendritic tree* to collect input signals from neighbouring cells at distributed locations. Output signals are conducted along an *axon fibre* before also branching out to multiple *axon terminals*. Along the fibre, conductivity can be improved by a so-called *myelin-sheath* that is produced by so-called *Schwann cells*.

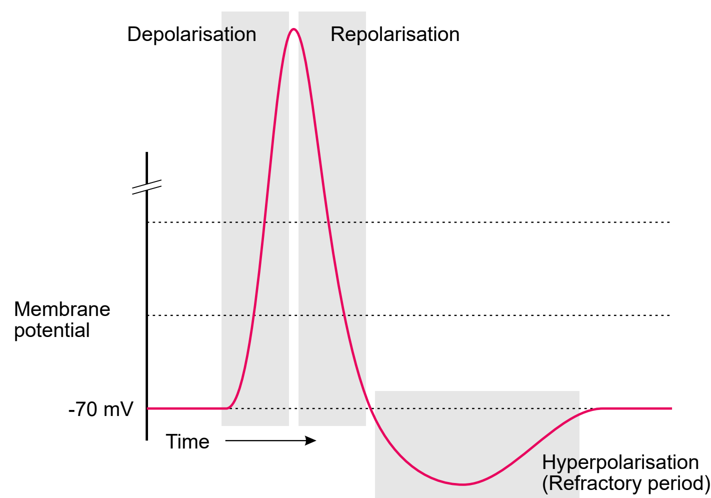


Figure 2.2: The typical shape of an action potential as described by the *Hodgkin-Huxley model* (Hodgkin et al. 1952); Figure taken from (Krol 2021). Stimulated by accumulated charges from external input currents, the membrane potential rises to a *depolarised* state, before falling via *repolarisation* into a *hyperpolarised* state, where the creation of further action potentials is blocked for a *refractory period* of time.

Generally, neurons exchange information through voltage *spikes* called *action potentials*. The typical course of an action potential is shown in Figure 2.2. It is created through voltage-controlled molecular electric processes of the involved channel proteins and usually triggered by the membrane potential crossing a particular threshold value through synaptic stimulus. The most accurate model of this process was developed by (Hodgkin et al. 1952) and is called the *Hodgkin-Huxley model*. As the shape of such action potentials does not vary a lot between incidences, information is only conveyed through the timing, frequency and spatial distribution of their occurrence.

During conduction across an axon fibre, the action potential is continuously refreshed along its path from the local membrane. However, when the axon is myelinated, the refreshing is restrained to the *nodes of Ranvier* between the *Schwann cells*.

More details and models on the structure and function of neurons and their membrane can be found in Sections 2.1 and 2.2.1 of (Petrovici 2016).

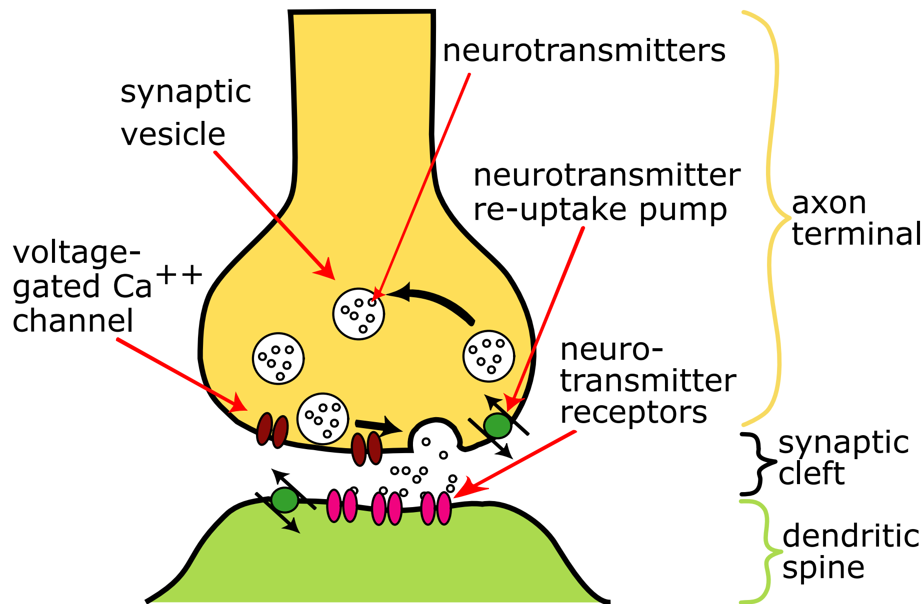


Figure 2.3: Schematic drawing of a chemical synapse; modified from (WikimediaUser 2006). An increased sodium concentration in the axon terminal, caused by an arriving *action potential* opening the respective *gated diffusion channels*, initiates the release of *neurotransmitter molecules* from *synaptic vesicles*. After diffusing through the *synaptic cleft*, they are detected by specific *receptors* at the *dendritic spine* of a neighbouring neuron. These receptors, being chemically gated ion channels, cause a change in the postsynaptic potential. After some time, the neurotransmitters are recycled by special *pumping channels* at the axon terminal.

2.1.2 Synapses

The points, where neighbouring neurons exchange information about occurred action potentials are generally called synapses. Mostly they are formed between an axon terminal of the presynaptic neuron and a dendritic spine of the postsynaptic neuron. Basically, there are two types of synapses; fast electrical ones and relatively slow chemical ones. While electrical synapses directly exchange ions through dedicated voltage gated diffusion channels, chemical synapses exchange special neurotransmitter molecules which cause an ionic current at the receiving dendrite.

Figure 2.3 shows the schematic process of a chemical synapse transmitting an action potential. Basically the increased voltage of an incoming action potential causes the neuron to emit transmitter molecules into the synaptic cleft which are detected by receptor channels at the dendritic spine, causing the postsynaptic neuron to respectively change its own membrane potential by ion-exchange with the surrounding liquid. Depending on the type of neurotransmitters and receptor channels, the effect can be excitatory or inhibitory. Once the postsynaptic neuron has integrated enough charge from all its dendritic synapses, it will create an action potential on its own.

More context on models describing the function of synapses can be found in Sections 2.1.3 and 2.2.2 of (Petrovici 2016).

2.1.3 Synaptic Plasticity

The specific coupling strength of synapses, which is also often referred to as the *synaptic weight*, varies significantly between individual synapses and neurons as well as over different time scales. Therefore, it is commonly associated with learning and storage of information. The general process of weight variation is thereby referred to as *synaptic plasticity*. In biology this has many different aspects, which basically happen on three different timescales.

Short-term plasticity is the fastest form of coupling strength variation, which is related to the amount of available neurotransmitter molecules. When the presynaptic neuron is firing faster than it can recycle the emitted transmitter molecules, it runs out of supply and the transmission of subsequent action potentials will be less effective. Historically, this was described by the *Tsodyks-Markram mechanism* (Tsodyks et al. 1997). According to the time scale of chemical synapses this type of plasticity happens on the order of milliseconds to seconds.

Long-term plasticity on the other hand is based on *Hebb's principle* (Hebb 1949) and happens in the order of minutes to hours. The physiological processes are not fully understood here, but the spike-timing-dependent plasticity (STDP) model (Bi et al. 1998; Markram et al. 1997) provides a mathematical description of the basic observation that neighbouring neurons firing in a correlated timing pattern will adapt their interaction strength according to the timing sequence of their respective firing activity. If the presynaptic neuron fires first (causal correlation), the synaptic strength will be increased. The other way round, if the postsynaptic neuron fires first (acausal correlation), the synaptic strength will be decreased.

On the largest time scale, in the order of days to years, neurons can even change their physical connection pattern which is called *structural plasticity*.

For more details on synaptic plasticity please refer to Section 2.2.2.2 of (Petrovici 2016).

2.2 Neuromorphic Computing

Neuromorphic computing generally aims to build computing systems that are based on the known aspects of how biological nervous systems work, which have been briefly summarised in Section 2.1 before. A review on the history and landscape of neuromorphic computing with a focus on large-scale systems is given in (S. Furber 2016). An extensive insight on the principles of neuromorphic computing and engineering is given in the books of (Ben Abdallah et al. 2022) from the engineering perspective and (Yu et al. 2017) from a more theoretical perspective.

2.2.1 Generations of Neural Network Models

For computational modelling, the biological neuron model, described in Section 2.1 is translated into a simple mathematical abstraction.

One can generally distinguish three generations of neuromorphic computing models. Although they all adhere to the same basic mathematical abstraction of a biological neuron, they differ in the kind of signals they process and the activation function gating the output signal.

2.2.1.1 Perceptron

The first generation of computational neuron models was called *Perceptron* and was developed in the late 1950s and early 1960s (Block 1962; Rosenblatt 1958). The Perceptron processes binary signals. Input signals are multiplied with real-valued weights and the activation function is a simple threshold. If the sum over the activated input weights exceeds the threshold value, the output is also activated. The output state is always propagated to the next unit at regular time intervals. The Perceptron model was one of the first attempts to learn how the cellular structure of the brain is connected to its cognitive and computational function. It could be seen that already this simple abstraction of biological neurons is able to show "interesting aspects of learning, discrimination, generalization, and memory" (Block 1962). However it was later shown that a single basic Perceptron unit is incapable of processing non-linear separation problems like for example an Exclusive OR (XOR)-gate does (Minsky et al. 1969). This can only be achieved by combining multiple of these units in a network of several layers, where signals propagate from an input layer over one or more hidden layers to an output layer.

2.2.1.2 Deep Learning Networks

The second generation is what is today called a conventional Artificial Neural Network (ANN). It processes real-valued signals and gates them with activation functions having a continuous codomain. In addition to this refinement of the neuron model itself, there was also a significant development in the network structure towards deeper networks with more hidden layers. However, while a small Perceptron network with a few layers can still be optimised analytically by solving a set of equations, this method is no longer viable within the vastly increasing parameter space of deep ANNs. For this reason, the methods of error backpropagation and stochastic gradient descent were developed since the late 1960s (Amari 1967, 1993).
The concept of Convolutional Neural Networks (CNNs) is a regularised version of a general deep multilayer neural network and was introduced in the 1980s (K. Fukushima 1988; K. Fukushima and Miyake 1982; K. Fukushima, Miyake, and Ito 1983; Kunihiko Fukushima 1980). In recent years these deep and convolutional neural network models have grown to incredible sizes, achieving great success milestones like e.g. finding new strategies in Go, one of the most complex board games of humanity trained by playing against another instance of itself (Silver, A. Huang, et al. 2016; Silver, Schrittwieser, et al. 2017), or e.g. Chat GPT[1] generating text in perfect human language, trained by feeding huge amounts of text input from the publicly available internet.

2.2.1.3 Spiking Neural Networks

In contrast to to the previously described computational neuron models, Spiking Neural Networks (SNNs) are modelled more analogueous to biology. As the name already tells, SNNs work on spike trains as signal type. A spiking neuron unit accumulates input spikes and emits an output spike when a threshold is exceeded. This is a substantial difference to the regular forwarding of an output signal in the non-spiking neural network models and introduces the concept of time to the model. Information can now either be coded in the mean spike rate or the precise timing of spikes.

[1] chat.openai.com/

Both coding schemes have their biological motivation as sensory input is mostly conveyed via precise temporal coding (cf. e.g. Reinagel et al. 2000 and Wehr et al. 2003) while muscle actuation is mainly signalled by average firing rates of motor neurons (Gerstner, Kreiter, et al. 1997). By ignoring the temporal structure of spike trains, rate coding is a simplification over the precise temporal coding, however making it robust against temporal noise on the input. Section 2.2.3 will focus on the statistical poisson properties of a rate-based coding.

2.2.2 Spiking Neuron Models

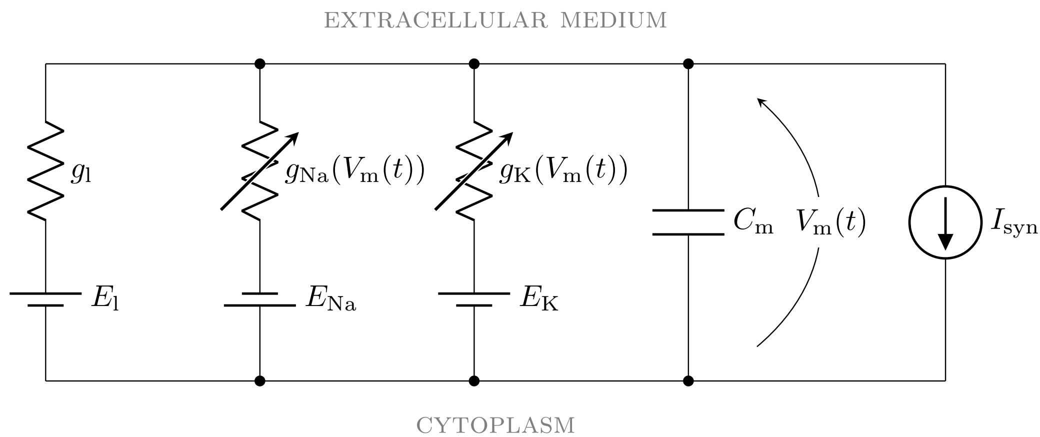


Figure 2.4: "Equivalent circuit of a Hodgkin-Huxley neuron, modelling the potential across the membrane. The conductances representing the sodium and potassium ion channels exhibit their own membrane-potential-dependent dynamics and modulate the respective currents." Synaptic input or external stimuli to the neuron are modelled by a current source onto the membrane.
Figure and caption text modified from Figure 2.3 in (Billaudelle 2022).

The most accurate spiking neuron model with respect to biological observations and data is the Hodgkin-Huxley model (Hodgkin et al. 1952) which was already mentioned in Section 2.1.1. It models the neuron membrane using an equivalent electric circuit (shown in Figure 2.4), charging a central capacitance through several parallel conductances. In the model, there is one static leak conductance and two dynamic conductance values modelling the voltage-dependent opening probabilities of sodium and potassium ion-channels. (Current-based) synaptic inputs or external (i.e. experimental) stimuli are incorporated in the model via an abstract current source to the membrane. In total, this is described by four differential equations, one for modelling the overall membrane dynamics and three for modelling the dynamics of three gating variables, describing the behaviour of the two modelled ion-channel types.

As this model is quite detailed and therefore complex, computational evaluation of large networks built from Hodgkin-Huxley neurons is very costly. The most simplified and therefore widely used spiking neuron model is the so-called Leaky Integrate-and-Fire (LIF) neuron which is even older than the model by Hodgkin and Huxley (Abbott 1999). The simplified equivalent circuit gets away with the dynamic conductances for modelling a spike and replaces them with a *manual* spike- and reset mechanism.

With this model, the membrane potential is modelled with a single differential equation given by

$$C_m \frac{dV_m}{dt} = g_l(E_l - V_m) + I(t) \quad . \tag{2.1}$$

Thereby $\tau_m = \frac{C_m}{g_l}$ represents the membrane time constant, quantifying the reaction speed of the membrane to the synaptic input $I(t)$, with C_m. When the membrane potential reaches a certain threshold V_{th}, the neuron emits a spike and goes to a reset potential V_{res} for the refractory period τ_{ref}:

$$\text{neuron spikes at } t = t_{\text{spike}} \Leftrightarrow V_m(t_{\text{spike}}) = V_{\text{th}} \tag{2.2}$$

$$V_m(t_{\text{spike}} < t \leq t_{\text{spike}} + \tau_{\text{ref}}) = V_{\text{res}} \tag{2.3}$$

In between these two models, there exist many different models, describing more complex neuron firing patterns like e.g. bursting or oscillations in more or less detail and accuracy. One of these models is the AdEx model (Gerstner and Brette 2009a), which has been implemented in analogue circuits on BSS-1 (Millner 2012) and BSS-2 ASICs (Aamir et al. 2018; Billaudelle 2022).
In all these models, as mentioned in Section 2.1.1, the shape of a spike event does not carry any information. Also the dynamics of their creation is not of relevance for the communication between neurons and synapses. Only the timing of where and when spikes are generated is important and has to be transmitted across an interconnect between emulated neuron circuits. This topic of spike event communication for large scale spiking neuromorphic computing will be discussed in more detail in Chapter 5.

2.2.3 Poisson Statistics of Spike Trains

Biological observations show a highly irregular timing of spikes in the cortex of the brain. As described before in Section 2.2.1.3, this irregular spiking behaviour can be interpreted with strong biological evidence as precise temporal coding of information. However sometimes a simplified model is desired and there also exist biological cases, where the precise temporal structure is ignored or lost due to noise, yielding a rate based coding.
The *independent spike hypothesis* (Dayan et al. 2001; Heeger 2000; Rieke et al. 1997) states that in a biologically motivated rate coding, the probability for a neuron spiking some time after a given initial spike only depends on an *instantaneous firing rate* $r(t)$, derived from the stimulus input rate. This characteristics can generally be modelled sufficiently with a poisson random process, where subsequent events occur randomly and independently of each other. (Gerstner and Kistler 2002; Stevens et al. 1995) even show that a poisson spike statistic can be formally derived from a LIF model with noisy input.
The work of (Heeger 2000) mathematically summarises the description of poisson statistics and proposes practical methods to generate such spike trains in a simulation environment. Especially, (Heeger 2000) presents extensions to the Poisson model in order to incorporate phenomena like the refractory period and bursting behaviour of biological neurons which turn out to violate the *independent spike hypothesis*. This is done by setting the instantaneous spike time to zero for the refractory period and interpreting the poisson spike train as an event stream and drawing a number of burst spikes from another (poisson) random process for each event.

With a constant Poisson spike rate r, the distribution of Inter Spike Intervals (ISIs) τ is described by an exponential function

$$p(\tau) = re^{-r\tau} \quad . \tag{2.4}$$

where $p(\tau)$ is the probability for a specific Inter Spike Interval (ISI). A derivation of this result can be found in Appendix A.2.1, which leans on the description given in (Heeger 2000).
The poisson statistics of spike trains will be used later in Section 6.2.2.6.

2.3 High Performance Interconnection Networks

As motivated before, efficient interconnection networks are essential for high performance computing in massively parallel supercomputers, as well as for global information exchange in the internet. Also large scale neuromorphic computing systems require efficient interconnection technologies in order to exchange spike events between the individual neuromorphic simulation or emulation cores, no matter whether they simulate the neural network digitally like SpiNNaker or emulate it in an analogue or hybrid approach like BrainScaleS. This Section will introduce the working principles and techniques on which high performance interconnection networks are based.

2.3.1 The OSI Model

Communication and Synchronisation between processing units or between distributed applications on several computers is a very complicated task with many challenges and requirements. For example, the communication is supposed to be efficient in terms of bandwidth and latency, and also secure against loss of data or malicious spying attacks. There might even be variable requirements with respect to Quality of Service (QoS), depending on the kind of transported data, meaning that some application might be fine with more latency than others, but in turn needs more bandwidth. For example a pre recorded video stream essentially needs bandwidth but does not care about latency, while on the other hand a video conference system requires low latency, but is probably more flexible with bandwidth, as the image quality can be dynamically scaled as feasible.
In order to enable the construction of interoperable computer networks fulfilling all possible current and future requirements, the Open Systems Interconnection (OSI) model was developed by the International Standardisation Organisation (ISO), starting in 1977 (Zimmermann 1980). The model was published as an international standard by the International Telecommunication Union (ITU) and ISO (cf. ITU 1994) and describes the communication process across different technical systems. It introduces seven layers of abstraction in the general network architecture. Each of these layers has its own special purpose and defines easily exchangeable protocols for the interoperation with the adjacent layers. A layer offers a set of services to the next upper layer and works on data from the layer below. Logically, components of each layer only communicate with other components on the same layer.
In the following paragraphs, a short summary of each of the seven layers will be given, explaining their purpose and the services, they offer.

- **L.1 Physical Layer:** The lowest layer, sometimes also called Bit-Transmission Layer deals with the encoding and transmission of elementary data symbols over a physical medium (e.g. mod-

ulated electromagnetic waves in electrical or optical cables or over the air). It may offer the maintenance of a connection across the physical medium in order to keep track of the encoding at the receiving side (e.g. by transmission of IDLE-characters while no actual data is transmitted). Another function provided by the physical layer is the multiplexing of multiple connections onto a single medium. The physical layer may also contain repeating elements in order to (re)-amplify the signal. By adding redundancy to the physical encoding, also error-detection or even -correction is already possible at this layer.

L.2 Data Link Layer: The purpose of the Data Link Layer is to reliably transmit data across one or more physical connections. This involves error-detection and -correction, which is achieved by adding redundancy checksums to the data stream, which is for this purpose divided into frames. Frames where errors are detected, but cannot be corrected are either directly requested for retransmission or reported to the Network Layer (L.3) for later retransmission.

L.3 Network Layer: The Network Layer provides a switching service on top of the Link Layer. While Physical (L.1) and Link Layer (L.2) work on point-to-point connections, the Network Layer now operates across the whole network and provides end-to-end connections and therefore provides network-level addresses. These connections can either be circuit switched or packet-based. In both cases, the Network Layer determines the appropriate route for the data towards the final destination as well as the forwarding of the data (packets) across one or more hops on that route through the network. On that way, the data stream has to be managed in order to avoid congested links and data loss because of resulting full buffers. For this purpose data packets can be further fragmented into so-called flow control units (flits). The concept of flow control will be explained later in Section 2.3.3.2.

L.4 Transport Layer: This layer is mainly an interface layer between the Network Layer (L.3) and the upper layers beginning with the Session Layer (L.5). It provides its own type of transport-addresses. Between a pair of transport addresses there can be more than one transport connection with individual requirements to Quality of Service. Packets that have errors either detected locally or reported up by lower layers have to be handled by this layer through requesting and granting of retransmissions from or to the remote end of the connection respectively. However, this error-handling might also be subject to quality of service requirements, as some applications might be fine with a limited error- or loss-rate. Another purpose of this layer is efficiently partitioning the data payload into packets for transmission across the Network Layer.

L.5 Session Layer: The Session Layer manages the communication on the level of processes. It implements services for the recovery of communication sessions after experiencing a broken connection.

L.6 Presentation Layer: This Layer is responsible for converting the system specific presentation format (e.g. ASCII or JPG) from the Application Layer (L.7) to an intermediate format. This may also include e.g. the conversion between Big-Endian and Little-Endian byte order on different systems. Thereby the Presentation Layer preserves the information content throughout

the transmission. It also handles compression and encryption of data in order to provide bandwidth efficiency and data security as these aspects are contained in the implementation details of the selected intermediate data format.

L.7 Application Layer: The uppermost OSI layer represents the user-applications that are using the communication infrastructure of the lower layers in order to exchange information. As the Presentation Layer (L.6) handles any conversions between the data representations used by the entities in this layer, the application entities do not need to be directly compatible. For example a chat application on a Linux system may easily communicate with another chat application on a Windows or Mac system as long as they all implement the same Presentation Layer protocol.

The following Sections will go into more detail on the design considerations at the Network Layer (L.3). Section 2.3.2 summarises the classification of interconnection networks. An overview on the switching aspect meaning the mechanisms directing the data on their route through the network is given in Section 2.3.3. Finally, Section 2.3.4 gives an insight into the considerations of how to best determine that route. For more details on the summarised topics the reader may refer to the book of Duato et al. 2003 which served as a basis for this overview Section.

2.3.2 Network Classification

Interconnection networks have been realised in a multitude of possible variants, each having their own advantages and disadvantages. Depending on the different constraints of the particular system they are used for, some aspects may be more important than others. One of the most important criteria to which interconnection networks are categorised is the network topology. Topology thereby describes the structure of the network graph, connecting the nodes. Based on this criterion, one can basically distinguish four classes of network topologies, which will be shortly described in the following paragraphs.

2.3.2.1 Shared-Medium Networks

As the name already tells, in these networks, the communication medium is shared among all nodes (cf. Section 1.5 in Duato et al. 2003). A great advantage of this structure is the inherent ability of broadcast and multicast communication, where one node sends information to multiple or all other nodes. However, at every time only one node is allowed to send data on the medium. This leads to the necessity of an arbitration mechanism between the nodes requesting access to the interconnect. Because of this time-multiplexed access scheme, such a network is not scalable, as the limited bandwidth of the shared medium will soon become a bottleneck when the number of nodes is increased. Figure 2.5 shows the connection scheme of a shared medium network, which is also called a *bus*.

Especially for the communication of multiple processors or functional units in a single system, bus networks can be much more complex, by sharing not a single data line, but having a large collection of different signal lines for data, address-information and separate lines for the arbitration mechanism. For example there could be individual `request` and `grant` lines for each node on the bus and a separate arbitration unit managing the right of access to the bus using these signals.

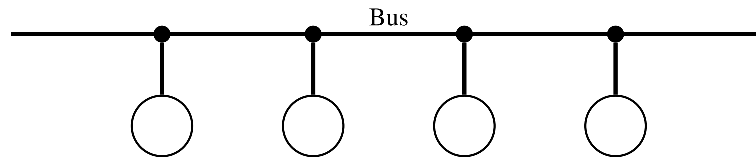


Figure 2.5: Graph of a shared medium bus-network

2.3.2.2 Direct Networks

In a direct network or *point-to-point network* (cf. Section 1.6 in Duato et al. 2003), nodes are directly connected to neighbouring nodes. Data messages are exchanged directly between the interconnected nodes and possibly have to move multiple steps through the network in order to reach their destination. Each node thereby contains a router unit, managing the connections to neighbouring nodes and the compute part of the local node itself. A connection thereby consists of an input channel and an output channel respectively and is commonly referred to as a *link*. This direct nature of connections leads to a good scalability of the network, as the overall bandwidth scales together with the number of nodes.
In a direct network, the nodes can be connected using different topologies. There are four main features that characterise a network topology. The first and probably most important one is the **node degree**, counting the number of links at every node. If all nodes in the network have the same degree, the network is said to be **regular**. The **network diameter** describes the maximum number of hops that is needed to route a message from a source node to a destination node. Last but not least, a network is called **symmetric** if it looks the same from each node. Generally, an irregular network will never be symmetric, but an asymmetric network can indeed be regular.

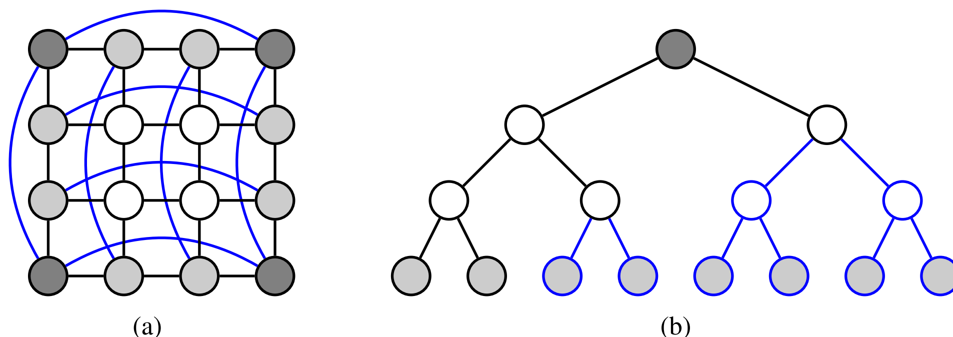


Figure 2.6: Topology Examples. **(a)** A 2D Grid, having a maximum node degree of four, is symmetric but not regular because of the corners (dark grey) and edges (light grey). When connecting the edges to each other (blue links) the network is called a torus, thereby becoming regular. **(b)** A (binary) tree is neither symmetric nor regular. When the distance between the root- (dark grey) and all leaf nodes (light grey) is the same, the tree is called balanced. However it still remains asymmetric and irregular.

Some example topologies are shown in Figure 2.6, showcasing symmetry and regularity in direct networks. A **2D grid**, as shown in Figure 2.6a with the black connections only, is characterised by a strictly orthogonal topology. The maximum node degree in a grid network is four, but it is only regular and symmetric excluding the edges with a degree of three and the corners with a degree of two. In contrast, a **2D torus** network is both regular and symmetric. It can be easily made from a 2D

grid through adding circular boundary connections (blue lines in Figure 2.6a). Thereby, the diameter is halved, making the torus network generally more scalable than a corresponding grid.
Figure 2.6b shows an example of a **binary tree** topology, either balanced or unbalanced. A tree is regular except for the root and leave nodes. As each node, except the root has exactly one parent, there are no cycles in a tree topology. It can also be advantageous for implementing some special algorithms (e.g. sorting) and synchronisation operations (e.g. a global barrier). The most important characteristic of a tree is that there is only one unique path between any two nodes. A disadvantage of this topology is that the bandwidth becomes a bottleneck near the root node, as all traffic destined to another branch has to pass the (sub-) root. Notably, every topology can be virtually transformed into a tree by removing connections.
Internally, the router unit in each node implements the services of the OSI Layers L.1 through L.3 and maybe also L.4. Especially the tasks of the Network Layer (L.3) are implemented here. The path of a message through the network is chosen by a **routing** algorithm (cf. Section 2.3.4) while a **switching** mechanism (cf. Section 2.3.3) rules the way in which messages are forwarded across its route, including physical channel usage, buffering and flow control.

2.3.2.3 Indirect Networks

In an indirect network (cf. Section 1.7 in Duato et al. 2003) each node has a simple network interface unit connecting it to an external switching device. Basically, the switching device takes the role of the internal router unit in direct networks, while connecting to other switches and endpoint nodes. Thereby, the computing nodes become less complex in design and are not restricted to specific network topologies through their node degree. Generally, direct and indirect networks are very similar with respect to their implementation. However, in the indirect case, messages have to take an additional step into and out of the network at the endpoint nodes, respectively.
An ideal switching topology is given, when messages from all input channels can simultaneously and asynchronously reach any free output channel. An example for such a **Crossbar** switch, realised by a matrix of atomic switching units is shown in Figure 2.7.

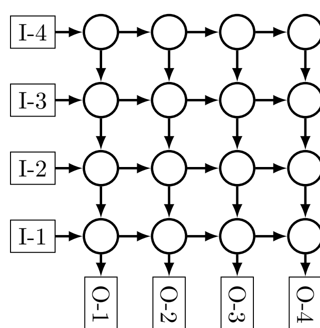
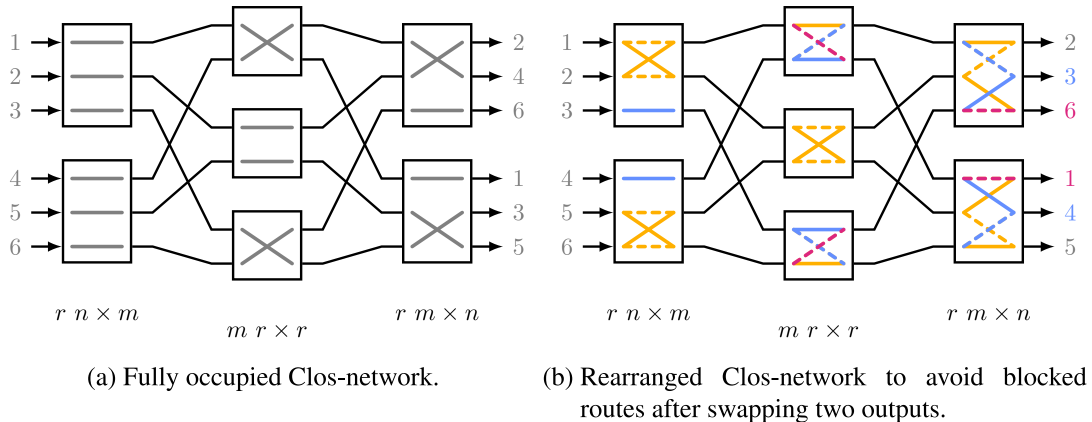


Figure 2.7: Schematic of a crossbar switch with $N = 4$ inports and $M = 4$ outports

However, for large networks a single big crossbar switch is not feasible, as it is limited by hardware constraints like e.g. the pin count of an integrated circuit. In direct networks, this is not a problem, as it is naturally partitioned to multiple distributed routers. Similarly, large indirect networks are implemented using multiple, ideally identical stages.
Multistage Interconnection Networks (MINs) are characterised by their blocking behaviour, as well

as their dimensions like the number of stages and the size and number of the individual crossbar switches in those stages. These characteristics will here be shortly explained at the example of a 6×6 **Clos-network** (Clos 1953; Lenfant 1978), which is shown in Figure 2.8.

In order to not be *blocking*, i.e. each input can be connected to every output simultaneously, a Clos-network is defined with three stages. The input stage features r switches with n inputs and m outputs, each connecting to m switches in the middle stage. These middle stage switches receive a connection from each switch at the input stage and in turn connect to each switch in the output stage. Last but not least, in the output stage again each switch receives a connection from every switch of the middle stage.



(a) Fully occupied Clos-network.

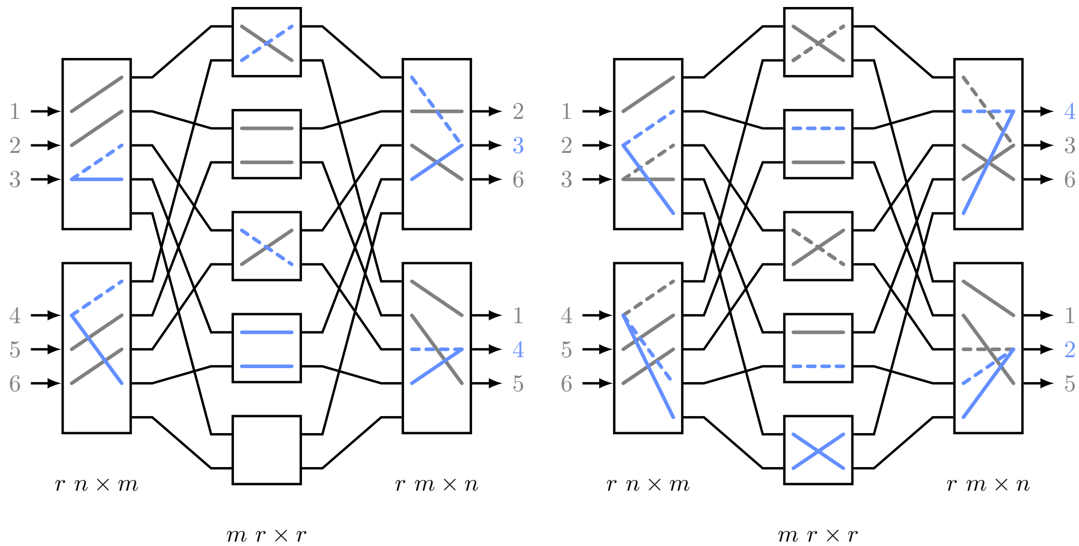
(b) Rearranged Clos-network to avoid blocked routes after swapping two outputs.

Figure 2.8: Rearrangably non-blocking Clos-network with $r = 2$ and $n = m = 3$

With this connection scheme it is possible to draw paths from each input to every output, as shown in Figure 2.8a. However, in this fully connected state it is not possible to swap two input-output connections without having to rearrange the other connections, as shown in Figure 2.8b. Swapping the output-mapping of **inputs** 3 **and** 4 will block the paths of **inputs** 1 **and** 6. To repair these paths, the **remaining paths** also have to be rearranged. Therefore this Clos-network is called *rearrangably non-blocking*. The dashed lines in Figure 2.8b indicate former connections while the solid ones indicate current connections.

In order to make the network completely *non-blocking* it needs to have at least $m \geq 2n - 1$ middle stage switches as shown in Figure 2.9. The initial connection pattern is the same as in Figure 2.8, but now there are two additional switches in the middle stage. When again swapping the output mapping of **inputs** 3 **and** 4, the new connections can now use one of the additional switches, leaving the **other connections** intact (Figure 2.9a). When swapping input 4 again with input 2, now also the second additional switch is used (Figure 2.9b). Another swap can now safely reuse any of the five middle switches.

Another popular MIN is a **Beneš-network** (Beneš 1965; Lenfant 1978), which is completely built out of 2×2 switches. For 2^n inputs there are $2n - 1$ stages with $\frac{n}{2}$ switches each. An n-input Beneš-network can be built recursively by duplicating an $\frac{n}{2}$-input network and adding an extra input- and output-layer connecting each input and output to both sub-networks respectively. A Beneš-network can also be seen as a Clos-network with $r = n = m = 2$ and is therefore also *rearrangably non-blocking*. It can be made completely *non-blocking* by using 4×4 switches and connecting the inputs and outputs twice to each sub-network. This is then called a 2-multi-Beneš network (Arora et al.



(a) Non-blockingly rerouted Clos-network after swapping two outputs.

(b) Non-blockingly rerouted Clos-network after a second swap of outputs.

Figure 2.9: Non-blocking Clos-network with $r = 2$, $n = 3$ and $m = 5$.

1990).

2.3.3 Network Switching

As stated earlier, *network switching* classifies the way in which messages are forwarded through the network. The following sections describe some key concepts of packet switched networks. Switching concepts include the arbitration of requesting input messages at the output ports (Section 2.3.3.1) as well as the location and amount of buffers on the data paths (Section 2.3.3.2). To avoid the risk of data loss due to overflowing buffers, flow control ensures that new data is only sent if the receiving side has enough buffer space available (Section 2.3.3.2). The amount of buffer space required in each switch or router unit for successful and efficient operation of a network, as well as the latency characteristics of message transmission largely depends on the applied switching strategy (Section 2.3.3.3). Depending on the switching- and buffering strategy, there can be problems like deadlocks and head-of-line blocking (Section 2.3.3.4) which can be addressed by virtual channels and virtual output queues (Section 2.3.3.5) respectively.

2.3.3.1 Scheduling

A crossbar switch (as depicted in Figure 2.7) receives messages on N input ports and forwards them to one or more of its M output ports. As only one inport is allowed to access an outport at a time, there has to be some sort of arbitration in case of conflict. Scheduling refers to this task of matching requesting inports to free outports. Generally, the scheduling algorithm should serve all inports equally under equal conditions and they should not influence each other. Also it should be simple and fast to implement in hardware for high message rates (cf. Section 1.6 in Philipp 2008).
The scheduling may be implemented in a central unit or parallelly distributed across the switching fabric. Another possibility is to implement a parallel scheduling that is distributed across the input and output ports.

A centralised scheduler needs to compute a new match between of requesting inports to free outports at every change in the request pattern. The computation of these matches can be complex and it is difficult to algorithmically guarantee the service requirements stated above.
In case of distributed arbitration, the requests travel horizontally on input rows in Figure 2.7 until they reach the intersection of their target output column. The switching points, intersecting inputs in rows and outputs in columns can basically be in one of three states (cf. Figure 1.10 in Duato et al. 2003):

1. The local input row is requesting access to this output column from the left side. The local request is granted and forwarded to the next row below. If the local input row also requests another column, the request is replicated to the right. A simultaneously forwarded request from the local column above is blocked.

2. A forwarded request from above is granted access to the local output below. The local input requests another column and is forwarded to the right.

3. A forwarded request from above is granted access to the local output below. The local input also requests this output column from the left and is blocked. However, a multicast request from the left may still be replicated to the right.

With parallel arbitration, the matching happens in three steps (cf. Section 1.6.2 in Philipp 2008):

1. Every inport signals a request to the outports corresponding to their current message.

2. Each outport implements a *grant arbiter* selecting one out of all incoming requests. If an inport requests more than one outport, it can also receive more than one grant.

3. The inport now either wants to replicate the message to all granted outputs in parallel (in case of a multicast message) or it needs to implement an *accept arbiter* selecting one grant at a time and replicating the message sequentially. However, if the message is not meant to be replicated at all, the *accept arbiter* has to reject all but the selected grant and thereby free the corresponding *grant arbiter* to select the next outstanding request.

The underlying problem of arbitrating a single resource can thereby be realised in different ways. A basic arbitration algorithm, called *RoundRobin* selects requesting inputs with rotating priorities. Based on this, there are many specific improvements and scheduling algorithms. One example for a scheduling algorithm, based on RoundRobin arbitration is the *iSLIP* algorithm (McKeown 1999). An extensive overview on existing arbitration and scheduling algorithms can be found in Section 1.6 of (Philipp 2008).

2.3.3.2 Buffering and flow control

In order to prevent corruption of data, each output port can only accept one input request at a time. If (in an $N \times N$ crossbar) there are multiple inports requesting a single outport, arbitration handles the selection of one of them, as explained in the last Section. All requests that have not been granted by the arbiter would be inevitably lost, unless the output port operates at a frequency that is N times higher than that of the input ports (cf. Section 3.2.1 of B. U. Geib 2012). Alternatively, the rejected

messages have to be buffered in the data path before the arbiter. In the latter case, their requests can be repeated until they will eventually be granted.
Buffers can be placed either at the inports or at the crosspoints interconnecting them. As placing buffers at the crosspoints needs a high number of N^2 small buffers, buffers are mostly placed at the input ports, requiring only N buffers. All incoming messages will be buffered there until they can be forwarded to the respective outport.
If messages arrive at an inport with a higher rate than they can be consumed by the outports, the buffers will eventually run full, again leading to data loss. To avoid this, backpressure has to be applied to the sending unit in order to control the rate at which messages arrive at the input buffers. This is described by the concept of flow control. For this purpose, messages are divided into flow control units (flits) on the switching level which are in turn subdivided into physical digits (phits) on the physical link as an atomic unit of data transfer. The capacity of all switching buffers must be an integer multiple of the flit size.
One way to implement flow control is to have dedicated control signals on a channel between two nodes or units. When the sender wants to transmit data to the receiver, it asserts a request signal and holds the data signals valid until the receiver in turn asserts an acknowledgement signal. The receiver must thereby only assert the acknowledgement if it can store the current flit into a buffer.
If the channel also transmits a clock signal, another possibility for flow control signals is to assert a stop signal just before the buffer becomes full. However the instant of time when the stop signal is asserted has to take the length of the channel into account with respect to transmission delay and clock frequency. When the receiver detects an almost full buffer condition, the buffer must have enough capacity left to store those flits that are already on the channel and will still be sent until the sender receives the stop signal.
Another way is to implement a **credit based flow control** (cf. Section 3.4.5 of B. U. Geib 2012). In this case the receiver initially notifies the sender about the number of flits it can store in its buffers. It is now the responsibility of the sender to maintain a credit counter and to only send out flits across the channel as long as it has credits. The receiver in turn has to restore the correct amount credits to the sender when buffer space is freed.

2.3.3.3 Switching Strategies

There are several basic strategies about how to guide a message towards its destination across the network. These strategies largely differ in terms of message latency and the rate at which messages can be sent. The switching strategy also has a large influence on the mutual blocking of different messages while traversing the network. For more details please refer to Section 2.3 of (Duato et al. 2003).
Circuit Switching first reserves a physical route through the network and afterwards streams the data through that route. For this purpose, a *routing header* is sent through the network preparing the data path towards the destination node. When the header has arrived, the destination node responds with an *acknowledgement notification*. When the source node receives this acknowledgement, the data transfer is started, finally freeing the route for other transfers. This approach requires no data buffers but induces a high setup overhead making this approach only feasible for long messages that occur only infrequently. Also, reserved data paths could severely block the setup of other message

paths.
On the one hand, the setup latency is proportional to the distance between source and destination in terms of hops through the network and scales with the latency of a routing decision as well as the transfer latencies between nodes, the latter impacting twice because of the acknowledgement notification. On the other hand, the data transmission-latency is proportional to the message length.
Packet Switching or **Store-and-Forward Switching (SAF)** resolves the issue of blocked paths by dividing messages into packets. Each packet is prepended by a header section containing destination and control information, which is used to make the routing decisions. The packets are individually routed across the network and completely buffered at each hop, introducing large buffer requirements. The transmission latency scales similarly as for *Circuit Switching*.
Virtual Cut-Through Switching (VCT) aims to reduce the transmission latency by not necessarily buffering the full packet before starting the routing decision and forwarding the packet. Instead, the routing decision is made as soon as the packet header is available and data flits are only buffered if the respective outport is blocked. Thereby packets are pipelined through the router units and cut through the output channels. This approach does not reduce the buffer requirement, as packets are still fully buffered in case of a blocked output port. However, it greatly improves the latency, removing the proportionality of the payload transmission latency to the transmission distance.
Wormhole Switching (WHS) addresses the problem of having to buffer complete packets in the routers. Now the flits of a packet will be stalled by flow control. Thereby, the packet is now buffered across several routing units and outports in the network. As individual flits don't contain the routing information of their packet they must not be intersected by flits of another packet. This implies that a blocked outport in one router can also cause passive blockage to other packets impacting the now blocked packet on another router unit. An example of Wormhole Switching is shown in Figure 2.10.

2.3.3.4 Blocking of Messages

Messages can be blocked at different places in an interconnection network. Outports will always block messages if multiple inports are requesting at the same time (cf. Section 2.3.3.1). Moreover, messages will be blocked if buffers run full at any point in the routing unit (cf. Section 2.3.3.2). Generally blocking of messages will cause increased overall transport latencies and decreased channel bandwidths. Some blocking situations are particularly bad, as messages are trapped in the network and will never reach their destination. For more details, cf. Chapter 3 of (Duato et al. 2003).
Starvation is a kind of message blocking, where an outport never grants access to specific inports. This is mostly due to an incorrect design or implementation of the scheduling algorithm and should be prevented in any case.
A **deadlock** occurs if multiple messages have a cyclic dependency across several network nodes, mutually blocking each other in their requests. Deadlocks can be dealt with either through prevention and avoidance by design or detecting and resolving them by dropping of packets. In particular, deadlocks can be avoided by choosing an appropriate routing algorithm and providing virtual channels (cf. Section 2.3.3.5).
A **livelock** situation occurs when a message is routed around the destination for ever and never reaching it. This can be avoided by restricting the number of routing decisions offside the minimal path towards the destination or exclusively using minimal paths.

Last but not least, **Head of Line Blocking (HOL)** occurs when an inport buffer contains two or more packets, where the second packet cannot advance because the first one is blocked by its outport. However, the second message actually wants to request another unblocked outport but cannot overtake the first packet due to the FIFO nature of the inport buffer.

2.3.3.5 Virtual Buffering

Figure 2.10 shows an example of a wormhole switched network with an exemplary deadlock and Head of Line Blocking. The messages **A**, **B**, **C** and **D** are cyclically blocking each other in their respective requests for an outport, thereby forming a *Deadlock* condition. Message **E** and **F** are blocked because their inport buffers are already occupied by a blocked packet. However these two messages actually do not want to request the same outports than the ones in front of them (messages C and E respectively). This situation is called *Head of Line Blocking*.

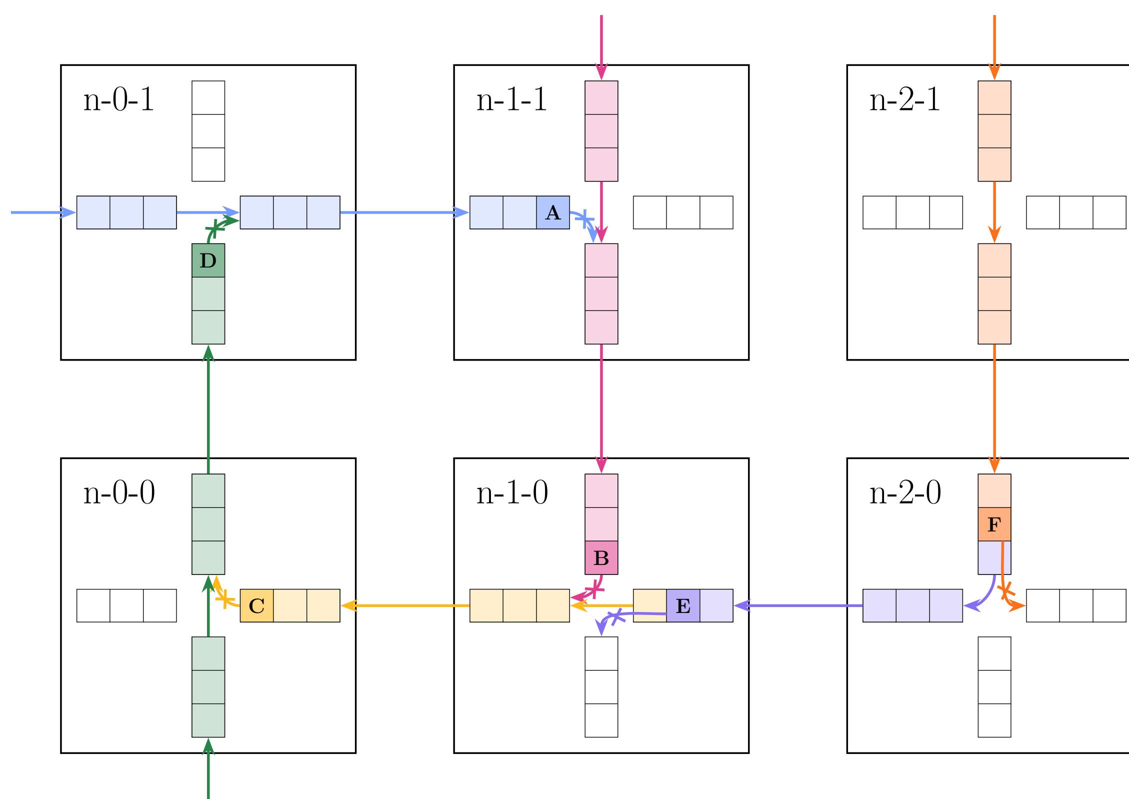


Figure 2.10: Example of a Deadlock and Head of Line Blocking in a wormhole switched network. For simplicity the depicted queues can represent inports and outports respectively. Darkly filled squares containing letters represent the packets' header flits, while the other coloured squares represent the data flits.

The impact of both blocking conditions, deadlocks and Head of Line Blocking, can be reduced by parallelised buffering at the places where the blocking occurs. **Virtual Channels (VCs)** (cf. Section 2.4 of Duato et al. 2003) multiplex the transmission of messages across a physical channel by allocating a pair of separate flit buffers at both ends of the channel. Now messages that were blocked by another message on the same outport can use another free VC and bypass the first message. A drawback in the use of VCs is that each VC needs its own flow control which also increases the traffic of acknowledgements and credits across the physical channel. Also the bandwidth of the

physical channel is now shared among all virtual channels which reduces the bandwidth and thereby increases the latency for single messages. Also the increased design complexity may have an impact on the message latency.
HOL blocking is also reduced, but not completely resolved by the introduction of virtual channels. To address this problem, the concept of Virtual Output Queues (VOQs) is introduced. It sorts incoming packets into different queues, corresponding to the requested outport at the inports. However, this is not quite scalable, as it requires N^2 buffers for a $N \times N$ crossbar.

2.3.4 Network Routing

An important aspect of interconnection networks is the choice of a specific routing algorithm, as it determines the path that a message packet takes across the network. As the design space of different routing algorithms is quite large and already covered extensively in Section 4.1 of (Duato et al. 2003), this Section will focus on information, necessary to understand the capabilities of the EXTOLL crossbar design, which are summarised later in Section 4.1.3.
First, a routing algorithm can be classified by the number of simultaneously addressable destinations. If messages can only be addressed to a single destination, the routing is referred to as *unicast*, while *multicast* messages may target multiple destinations simultaneously. This can either be implemented by replicating messages sequentially at the source node, or distributing it to the respective directions at intermediate steps through the network. The latter is more scalable, as it only creates additional traffic where it becomes inevitable and the message is injected to the network only once, instead of once per destination and thereby freeing the sending node immediately. When a message targets all nodes in the network, it is referred to as a *broadcast* message.
Furthermore, a routing algorithm may be either *deterministic* or *adaptive*. Deterministic routing algorithms on the one hand will always produce the exact same path for a given pair of source and destination. On the other hand, adaptive routing algorithms may choose directions between several alternatives or even without any constraints, depending on the level of congestion or blockage of output ports. While a deterministic algorithm can be particularly designed to be deadlock free, i.e. without cyclic dependencies, this is not so easy for adaptive algorithms, as their decision will depend on the network state. Also, packets routed deterministically will always arrive in the same order as they have been injected, while adaptively routed packets may overtake each other, thereby requiring re-ordering at the destination.
Finally, routing algorithms may be implemented either using a *lookup table* or a *Finite-State Machine (FSM)*. An example for the latter method would be dimension-order routing, e.g. in a grid or torus network, where messages are forwarded linearly in one direction until the respective coordinate matches that of the destination node. While this is quite efficient in its implementation, it is however mostly restricted to specific topologies and therefore not considered flexible. Alternatively, a table-based implementation offers the full flexibility with respect to supported topologies, but comes at the cost of extensive memory requirements. The memory footprint of table-based routing can be reduced by a hierarchical approach where the network is partitioned into multiple sub-networks. Messages are first routed to the destination region and finally inside that region to the particular destination node. The partitioned lookup-tables can thereby be accessed in parallel to optimise the time needed for the routing decision.

3 The BrainScaleS-2 System

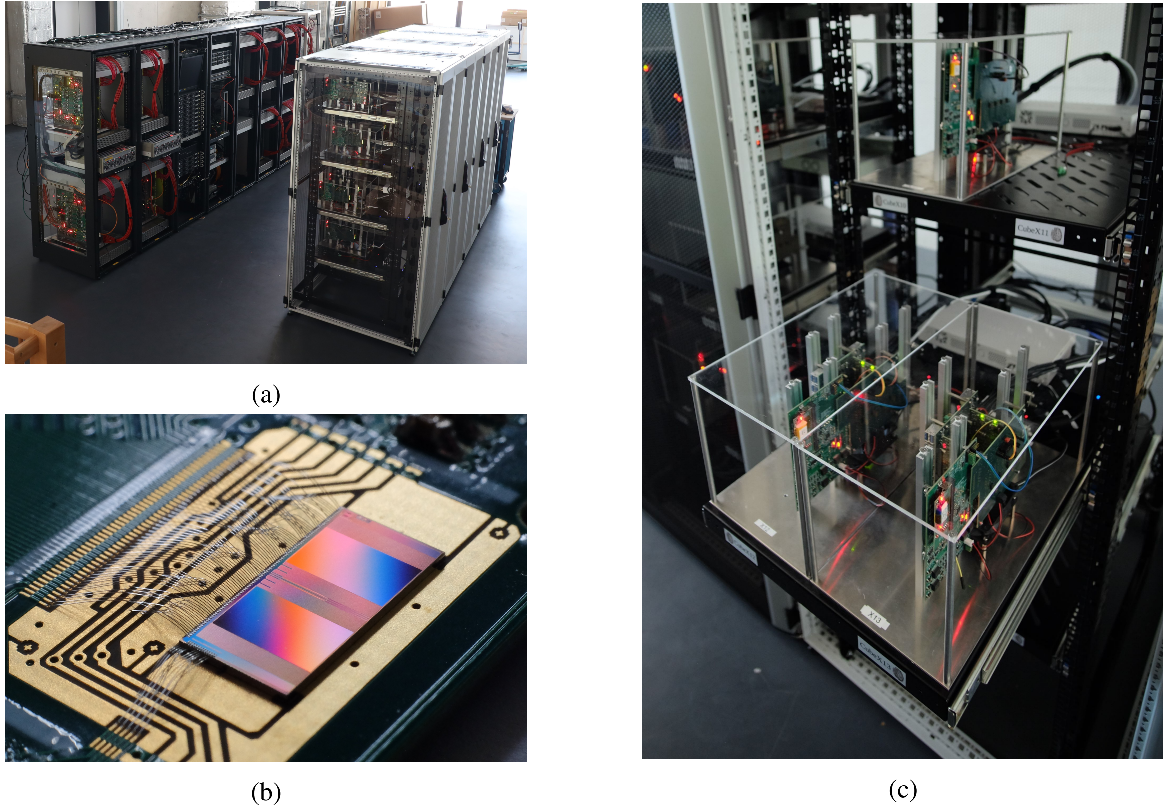


Figure 3.1: **(a)** Photograph of the BSS-1 (on the left in black) and BSS-2 (on the right in white) systems in the experiment hall at the European Institute for Neormorphic Computing (EINC) in Heidelberg. Courtesy of Björn Kindler. **(b)** Photograph of a BSS-2 HICANN-X chip, wire-bonded to a carrier Printed Circuit Board (PCB). Courtesy of Eric Müller.
(c) Photograph of multiple BSS-2 hardware setups, available for neuromorphic experiment execution. Each setup hosts two independent HICANN-X ASICs, hidden under a protection cap between the glowing red LEDs. One FPGA is used per ASIC to bridge the communication gap towards a host-computer. Courtesy of Eric Müller.

Figure 3.1a shows a photograph of both the BSS-1 wafer-scale system as well as the BSS-2 single-chip system. Both systems were recently moved to the new EINC[2] building in Heidelberg.
This Chapter focuses on the HICANN-X ASIC design (Section 3.1) and the communication FPGA (Section 3.2) as well as the software stack supporting the BSS-2 system operation, as they are used and enhanced respectively in Part III of this thesis.
Figure 3.1c shows a photograph of some BSS-2 hardware setups. These setups, which are called *Cube Setups*, have originally been developed for the purpose of prototyping the BSS-1 wafer mod-

[2] www.einc.eu

ule and have been modified to be used with the HICANN-X chip (Güttler 2017; Kleider 2017; Schreiber 2021). Each of these setups hosts two independent HICANN-X chips and four communication FPGAs, two of which are connected to one of the chips respectively. These FPGAs control the realtime experiment flow and are connected to a cluster of host computers via 1 $\frac{\text{Gbit}}{\text{s}}$ Ethernet connections. A photograph of a HICANN-X ASIC with wire-bond connections to a carrier PCB is presented in Figure 3.1b.

A report on the current state of the BSS-2 System development and tutorial applications has recently been published in (Müller, Emmel, et al. 2023).

3.1 The HICANN-X ASIC

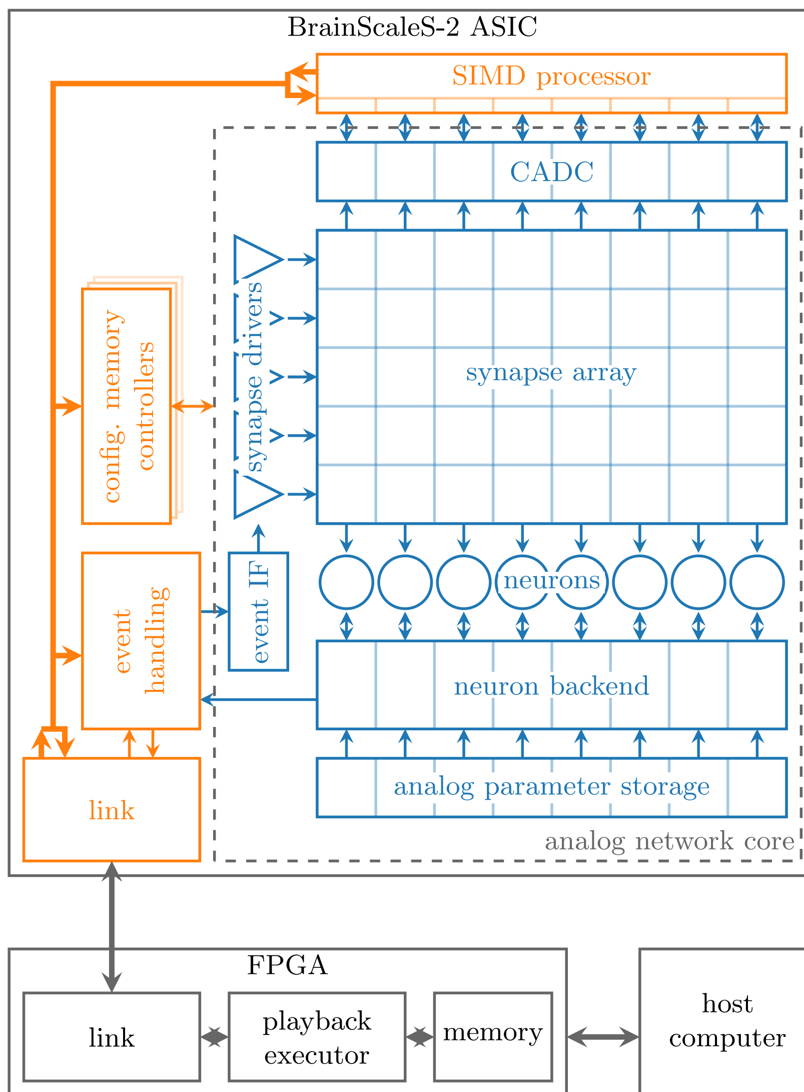


Figure 3.2: Block diagram of a BSS-2 ASIC design and its interface periphery; taken from (Müller, Arnold, et al. 2022). The analogue network core, depicted in blue represents one of two mostly symmetric hemispheres visible in Figure 3.1b. The digital blocks, coloured in orange encompasses one of two Plasticity Processing Units (PPUs) as well as the digital event handling, memory controllers and the serial high-speed links connecting to the periphery.

An overview block diagram of the HICANN-X design can be viewed in Figure 3.2. In this Section the features and function of the most important building blocks on the BSS-2 HICANN-X ASIC will be shortly summarised. The sequence of this summary orients at the overview and explanations given in Chapter 3 of (Billaudelle 2022). For more detailed information on the various features and circuits of the BSS-2 neuromorphic chips, the reader may refer to the work of (Billaudelle 2022) and the original publications cited alongside the respective paragraphs.

3.1.1 The ANNCORE

The HICANN-X chip features two symmetric hemispheres, each containing 256 neuron circuits and as many rows of synapses interfacing them. The design is based on the so-called Analog Neural Network Core (ANNCORE), containing the analogue circuits emulating the accelerated neuron and synapse dynamics.

Spike events are inserted into the synapse array with a 14 bit address label through synapse driver circuits, described in (Billaudelle 2017). These will inject received events directly into the connected row of synapses by providing precisely timed control signals and forwarding a 6 bit event address which is compared to locally programmable address labels to the synapse circuits. Synapse drivers themselves are addressed by a 5 bit address, also extracted from the spike label.

The synapse circuits, originally proposed by (Friedmann et al. 2017) are the central building blocks of the BSS-2 neuromorphic ASICs. They define the topology of the emulated neural network by interfacing the neuron circuits to input spikes from other neurons. To this purpose, they generate input current pulses of a certain width and strength for the connected neuron circuits. The width of these pulses is controlled by the synapse driver circuit and can be modulated to implement a simplified version of the short-term plasticity (STP) model proposed by (Tsodyks et al. 1997). The pulse strength on the other hand is determined by a configurable 6 bit weight value, stored in an SRAM local to each synapse. The synapses also implement sensor circuits to measure the causal and acausal correlations of pre- and postsynaptic spike events. This enables the implementation of accelerated STDP learning rules (cf. Section 2.1.3) using the two on-chip Plasticity Processing Units (PPUs) (Friedmann 2013; Friedmann et al. 2017).

The neuron circuits on the HICANN-X implement an extension of the LIF model (AdEx) and have been described by (Aamir et al. 2018; Billaudelle 2022). In order to emulate larger neurons, either for the purpose of more synaptic inputs or for modelling structured neurons with multiple compartments, multiple neuron circuits can be combined using programmable conductances (Kaiser et al. 2022). The digital neuron backend logic, described by (Kiene 2017) is responsible for creating new spike events and deriving timing signals like the refractory period. A large parameter configuration space allows the neurons to be tuned to various target dynamics and to be calibrated against production mismatch between individual instantiations and chips.

All analogue circuits on the chip are highly parametrisable through local SRAM and a vast amount of Digital to Analog Converters (DACs). Analogously, analogue signal traces (as e.g. membrane voltage- or conductance traces) can be digitised using a high speed Membrane ADC (MADC). Additionally there are two arrays of Column-parallel ADCs (CADCs) providing 512 channels of parallel access to analogue values from the correlation sensors in the synapse array as well as the neurons' membrane voltages to the PPUs. This massively parallel readout comes at the cost of lower

resolution and sample rates as compared to the MADC (Schreiber 2021).
An analogue readout chain allows to connect various internal voltage states directly to two output channels (Kiene 2017). These can be either directly connected to i/o-pads of the chip or digitised using the MADC (Billaudelle 2022).

3.1.2 The Digital Part

The PPUs have been successfully used for implementing accelerated learning (Bohnstingl et al. 2019; Wunderlich et al. 2019) and embedded cybernetics (Schreiber 2021). Both processors have access to a small amount of private local SRAM memory. Data and instructions can also be fetched block-wise from a larger region of shared memory, transparently provided by the communication FPGA (Pehle 2021). Through the chip's communication bus, the PPUs have full access to the complete configuration and state space of all components on the chip, as well as on the FPGA. Additionally, the PPUs feature a custom Single Instruction Multiple Data (SIMD) vector extension (Friedmann 2013; Friedmann et al. 2017). With this, the PPUs have access to the parallel configuration space of the synapse array and are connected to the Column-parallel ADCs.

		synapse driver top				synapse driver bottom				L1 → L2			
		0	1	2	3	0	1	2	3	0	1	2	3
neuron output channels left of anncore	0	X				X				X			
	1		X				X				X		
	2			X				X				X	
	3				X				X				X
neuron output channels right of anncore	0	X				X				X			
	1		X				X				X		
	2			X				X				X	
	3				X				X				X
L2 → L1	0	X	X	X	X	X	X	X	X	X			
	1	X	X	X	X	X	X	X	X		X		
	2	X	X	X	X	X	X	X	X			X	
	3	X	X	X	X	X	X	X	X				X
background generators	0	X								X			
	1		X								X		
	2			X								X	
	3				X								X
	4					X				X			
	5						X				X		
	6							X				X	
	7								X				X

Figure 3.3: Schematic connectivity matrix of the event routing crossbar in HICANN-X. Connections marked with an 'X' are actually implemented on the chip. This Figure was taken from (Spilger 2021).

Finally, event communication, either recursively inside the chip, or with the outside is coordinated through a digital event handling block. It implements a sparse and programmable crossbar matrix, a schematic view of which can be seen in Section 3.1.2. The input channels are thereby listed on the left side and output channels on the top respectively. Spike events originating from inside the chip are either generated by the neuron backend or by dedicated background spike generators, offering regular or poisson-shaped spike trains (Johannes Schemmel, Billaudelle, et al. 2022). Generally, each input and output path has four channels, of which each can handle one event every two clock cycles (Johannes Schemmel, Billaudelle, et al. 2022). In order to save precious hardware resources,

only those connections marked with an 'X' are implemented on the chip.

This on-chip routing fabric is generally referred to as Layer 1 (L1), while the Layer 2 (L2) connections refer to the external event transport through eight high speed serial links, connecting the HICANN-X chip with its periphery. While L1 events are always transported immediately without timing information, solely based on their address label, events on the L2 are annotated with timestamp information, which is used to counteract latency variations on their way across the serial links. Incoming events from the L2 are delayed until their timestamp plus a programmable delay value matches the internal system time (systime) counter before they are injected into the L1 router (Alexander Schmidt 2017). For this to work, the systime counter has to be synchronised with the communication FPGA (Rettig 2019a). This synchronisation mechanism will be discussed in detail in Section 7.2.1.

The serial links not only transport realtime data like spike events or MADC samples, but also configuration data and status read outs, as well as external memory accesses from both PPUs. In total, there are 10 independent data queues traversing the links towards the HICANN-X and 12 queues in the opposite direction. The transparent multiplexing and encoding of these independent queues across the physical links is handled by a universal translator (UT) module, developed in (Karasenko 2020). More details on the UT will be given in Appendix B.1 and Section 7.3.3. Together, the 8 serial links offer a total bandwidth of 8 $\frac{\text{Gbit}}{\text{s}}$ (cf. Karasenko 2020).

3.2 The Communication FPGA

In order to bridge the gap in latency and bandwidth required to operate neuromorphic experiments at a speed-up factor of 10^3 compared to biological realtime, an FPGA acts as realtime experiment host for the neuromorphic computing system. The host computer compiles so-called playback programs containing all the configuration data and stimulus events, required to run a neuromorphic experiment. This also includes instructions and data for executing programs on the PPUs. The playback programs are then executed by the fpga down to the neuromorphic chip. Once configured and started, the neuromorphic experiment will evolve on the chip, completely asynchronous to the fpga and host software stack. It is however possible, to interact with the evolving neural network emulation by collecting observation (trace) data or changing the configuration.

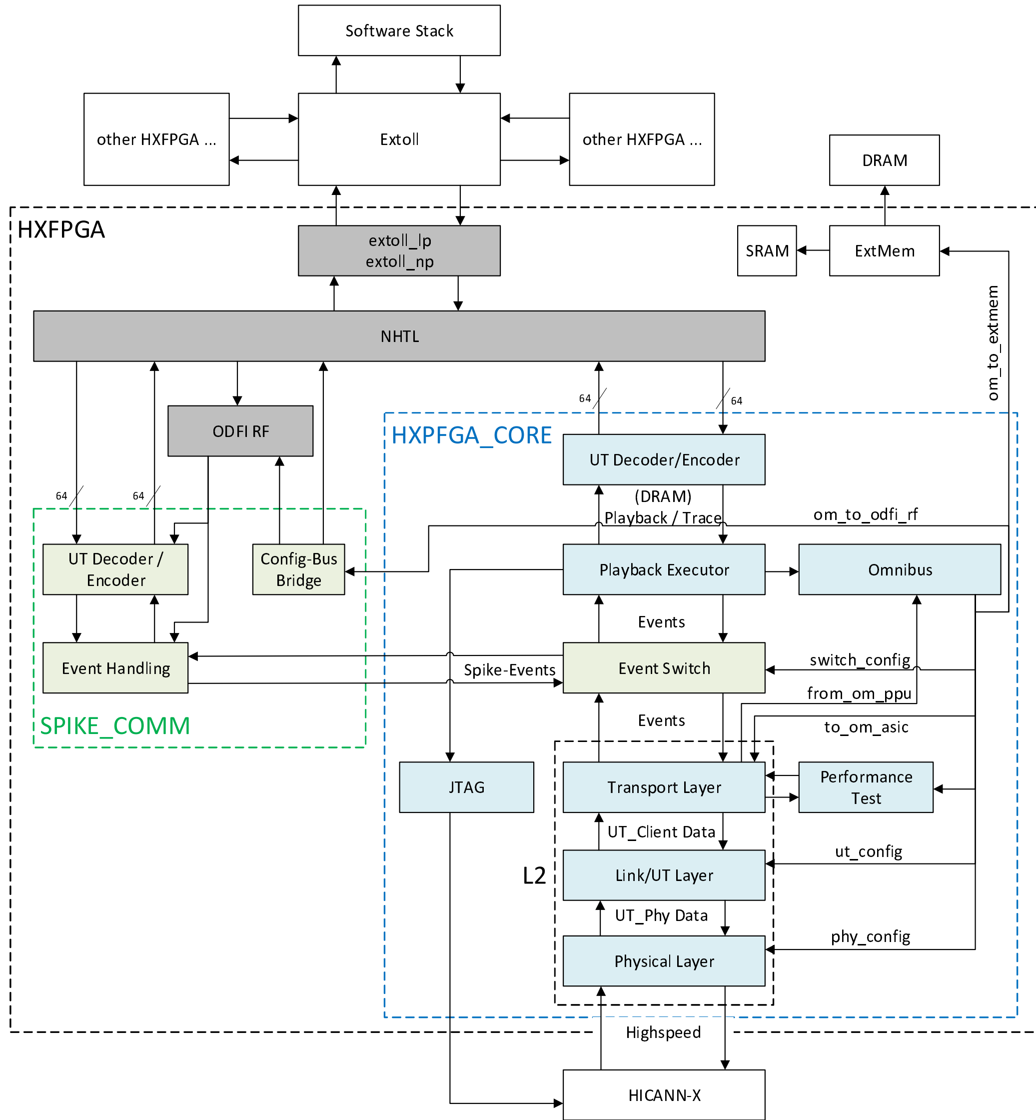


Figure 3.4: Block schematic of the BSS-2 communication FPGA; modified from (Rettig 2019a). Blue boxes show design blocks already present in the original HXFPGA_CORE, green boxes show design blocks developed in the scope of this thesis and grey boxes show design blocks adopted from (Thommes 2018).

A block schematic of the FPGA design is shown in Figure 3.4. The FPGA manages the communication with the HICANN-X chip as well as with the host computer, which is connected through a packet-based network interface. In the original system (cf. Rettig 2019a), the HXFPGA_core connects to an Ethernet network interface and implements an ARQ protocol (Karasenko 2014) in order to secure the connection against corruption and loss of transmitted data. The design which is subject to this thesis, however makes use of the EXTOLL network technology (cf. Chapter 4). With this, a higher bandwidth and lower latency is offered, as compared to Ethernet. This enables the low-latency communication of spike events across the network to other FPGA nodes alongside the playback and trace traffic.

The following Subsections will describe the purpose of each module block in the FPGA design, as depicted in Figure 3.4. The description will thereby start at the serial links from the chip at the bottom of the Figure and walk through the design towards the network interface at the top.

3.2.1 L2 Chip Communication

As already stated in Section 3.1.2 on page 37, the HICANN-X chip sends events to the outside world on the L2 connection. The BSS-2 FPGA design splits this functional block into three distinct parts roughly according to the OSI model (cf. Section 2.3.1).

The *Physical Layer* implements the sending and receiving serialiser modules necessary to drive the links towards the chip. These are basically a 65 nm version of the low voltage differential signaling (LVDS) serialisers, as already used in the BSS-1 chips (Scholze, Eisenreich, et al. 2011). According to (Karasenko 2020), the links implementation offers a physical data rate of 1 $\frac{\text{Gbit}}{\text{s}}$ at a serial clock-frequency of 500 MHz.

The *Link/UT Layer* performs the UT operation by accepting type-labelled data words at the client side and converting them into a continuous stream of encoded data for the physical layer. In the opposite direction, it decodes the received data stream and re-emits the previously tunnelled typed data from the chip. Additionally, the UT offers flow-control and data security services upon the non-blocking and unsecured physical transmission layer (cf. Karasenko 2020).

The *Transport Layer* handles the efficient distribution of incoming data across the 8 high speed links (Kanzleiter 2018) and also implements the transparent tunnelling of the FPGA's system configuration bus (*Omnibus*, Friedmann 2013, 2015) into the chips system bus and vice versa from the PPUs.

Besides this, the L2 functionality also includes a mechanism to synchronise the system clock counters between the chip and the FPGA. This is required to minimise the jitter of spike event latencies across the L2 links by delaying events with respect to their timestamp. The systime counters thereby need to be synchronised to be able to interpret the timestamp correctly on both sides of the link. This synchronisation mechanism has been introduced in (Rettig 2019a) and will be summarised in Section 7.2.1.

A configuration side channel to the chip is provided with a JTAG port. This is used to initialise the physical links at startup and for debugging purposes related to the function of the high speed links.

3.2.2 The Playback Executor

The purpose of the Playback Executor, as described in (Rettig 2019a) is to control the data transfer between the user and the chip, by demultiplexing data transfers from the host network interface to the different system units and multiplexing data responses from those units back onto the network interface. An important task of the executor is also the timed execution of these data transfers and synchronisation commands. This means that the user can exactly specify points in time, when specified stimulus spike events, configuration data or status readout commands shall be executed, relative to the execution of a special reset command. For this purpose, the executor receives a UT-decoded stream of index-data pairs from the network interface, which is called a playback program. The index thereby indicates the type of the instruction to be executed. The following instructions are currently supported by the executor:

1. **JTAG Instructions:**
   They control the JTAG port towards the chip and provide the data to be transmitted through this side channel for configuration and debug purposes.

2. **Timing Instructions:**
   These instructions trigger timing related operations including
   - **Wait-until:**
     blocks the execution of the next instruction until a sleep counter has reached a specified value. Alternatively there are wait instructions to wait until the JTAG or Omnibus are idle. In the scope of this thesis, global barrier and interrupt instructions where added, for global synchronisation in the EXTOLL network (cf. Section 7.2.2).
   - **Timer Reset:**
     resets the afore mentioned sleep counter to zero.
   - **Systime Init:**
     triggers the synchronisation of the systime counters in the ASIC and the FPGA. Thereby the FPGA will take over the current systime value from the chip. It is currently not possible to have the ASIC take over the systime value from the FPGA.
3. **System Reset:**
   Resets all modules downstream of the executor and asserts the reset pin at the HICANN-X ASIC.
4. **Spike Events to L2:**
   These instructions insert either one, two or three spike events into the L2 link network. These are transported as stimulus spikes to the chip and at this point only contain the 14 bit spike address label. The executor will attach the lower bits of the current systime value as a timestamp to each event.
5. **Omnibus Instructions:**
   They resemble read and write commands to the system configuration bus, the Omnibus. These instructions contain the target address for the desired access and whether it is a read- or a write-command. In case of a write command, the payload data follows directly after the address in another instruction word.

In order to prevent frequent stalling of the network interface each time a downstream unit blocks the execution due to congestion or simply at every wait-until operation, playback programs received from the host are buffered in a dedicated memory in the FPGA which is also extended by external DRAM.

3.2.3 Spike Event Communication

In order to communicate spike events between separate BSS-2 ASIC-FPGA pairs and thereby largely extend the emulateable model size, the main goal of this thesis is, to develop an efficient and reliable way to transport spike events through a packet-based network. This functionality is implemented in the *SPIKE_COMM* core, shown with green blocks in Figure 3.4. For more detailed information about the implementation of the spike communication block, confer to Chapter 7.
At first the event stream from the L2 to the *Executor* and vice versa has to be interjected. The *Event Switch* (cf. Section 7.1) implements this functionality by replicating the received event stream from

the chip to the *SPIKE_COMM*. Spike events received from the *SPIKE_COMM* are forwarded to the L2 in parallel to those from the *Playback Executor* and are merged by the existing transport layer *L2 switch* which was introduced by (Kanzleiter 2018). This *L2 Switch* operates on a UT interface (cf. Appendix B.1) and distributes transfer data items evenly from multiple input- to multiple output interfaces. The distributing hardware design is organised as a matrix of modular shifter units, where inputs are assigned to rows while outputs are assigned columns. New items are alternately accepted either from the current row, or the neighbouring row above. Buffered items will alternately be forwarded either to the neighbouring column to the right or down the current column towards an output interface. (Kanzleiter 2018)

The *SPIKE_COMM* block includes a destination lookup (cf. Section 7.3.2) indexed by the source spike label and mapping to a destination network address as well as another destination spike label. After this lookup operation, events are aggregated to form larger packets (cf. Section 7.3.3), as they are addressed to the same destination network node. This aggregation is necessary to minimise the share of bandwidth required to transmit the mandatory network header of each packet compared to the actual event payload (cf. Section 5.6.1). This introduces a trade-off between latency caused by the aggregation process and limits the bandwidth efficiency due to the so-called header overhead (cf. Equation (5.2)). Another point of optimisation for this trade-off is the packet-rate inserted to the network, which is also a limiting factor.

The *UT Encoder / Decoder* (Karasenko 2020) in this case encodes the number of parallel events accepted at the input of the accumulation buffer. This could in the future also encode different basic types of data exchanged between BSS-2 systems aside of spike events.

Together with the *Playback Executor* (Section 3.2.2) this block forms an Application Layer in terms of the OSI model (Section 2.3.1). However, one can argue that the network addresses provided by the routing lookup (cf. Section 7.3.2 and Section 7.3.3) already represent a part of the Network layer.

3.2.4 The EXTOLL Interface

As motivated in Chapter 4 on page 48, the EXTOLL network technology is used in the scope of this thesis to connect multiple BSS-2 systems and transport the host-communication, as well as the neuromorphic spike event communication between them. The EXTOLL Network (cf. Chapter 4) is interfaced to the BSS-2 FPGA design by basically four layers that can be roughly matched with the according OSI layers.

The NHTL, developed in (Thommes 2018) acts as Transport Layer towards the network. It accepts data words from the Application Layer, i.e. the `Playback Executor` and the `SPIKE_COMM` partition. It then packs these inputs to packets with a network header according to the network layer protocol. Data from the executor is sent to a previously configured partner host, while spike event data is sent to the destination indicated by the data itself.

The Network- and Link Layer are provided by the EXTOLL Network-Port (NP)- and Link-Port (LP) IP modules respectively. The Physical Layer (not shown in Figure 3.4) is implemented by GTX transceivers, which are hard-IP blocks on the FPGA fabric, provided by the FPGA vendor (cf. Xilinx Inc. 2018). These are specifically parameterized to be compatible with the EXTOLL network. Notably, the LP provides a global barrier and interrupt mechanism (Burkhardt 2007, 2012), which is used in the scope of this thesis for the purpose of synchronising the systime counter between

multiple BSS-2 FPGAs (cf. Section 4.1.2 and Section 7.2.2).

3.2.5 Omnibus and ODFI Registerfile

In this Section the used configuration bus technologies will be briefly introduced. As the FPGA design has to interface both the BSS-2 neuromorphic chip, as well as the EXTOLL network interface IP, two different buses are implemented. The core parts of the FPGA design and the HICANN-X components are configured via the Omnibus fabric, which was introduced by (Friedmann 2013, 2015). However, the EXTOLL network infrastructure and for compatibility reasons also the network interface units integrated into the FPGA design, have their own configuration registerfile architecture (Computer Architecture Group 2018). As the EXTOLL registerfile offers global network access to the local configuration space, this is also used for the configurations related to the spike communication architecture (cf. Section 7.6). However, as the BrainScaleS experiment flow is based on the *Playback Executor* and its Omnibus interface, some bus bridge is needed in order to integrate the network configuration space into the experiment configuration space (cf. Section 7.6.4).

3.2.5.1 The Omnibus

The original design intent for the Omnibus was to provide a communication bus between the PPUs and all modules on the BSS-2 ASIC for configuration of all kinds of parameters, as well as instruction and data fetching from external memory. Therefore it seems an efficient design decision, to also incorporate the same bus architecture into the FPGA design, to make both core system parts natively compatible to each other with no need for protocol conversion. This enables transparent master access from playback programs directly to the chip configuration space and native cache fetching operations from the PPUs to the external memory attached to the FPGA.
The Omnibus interface is designed in accordance to the Open Core Protocol (OCP) specification (OCP 2009). According to (Friedmann 2013), the implementation at hand features a 32 bit address space with 32 bit data words that can be enabled byte-wise. Both read and write accesses have to be acknowledged to the requesting master by the respective slave. As the bus has to span long physical ranges on the chip (in the order of multiple millimetres) and even a chip to fpga tunnel, it is pipelined and can accept multiple requests, before the first response is acknowledged. The order of access instructions is thereby always guaranteed to be fixed. Flowcontrol is ensured by implementing a handshake protocol in the request- as well as the response path. The bus system supports multiple masters with fixed priority arbitration and spans a tree through the address space towards the connected slave units. There are interface modules for register targets, (de-)serialisers, as well as for RAM blocks. As the coding of a complete bus tree in an HDL would be rather tedious, the implementation uses code generation with macros defined in the M4 language (Kernighan et al. 1977; Seindal et al. 2021) which will then generate an according SystemVerilog (SystemVerilog 2004) module.

3.2.5.2 The ODFI Registerfile

Configuration and status registers in the EXTOLL network hardware and FPGA IP are generated using the ODFI / CAG register file generator (RFG) (Computer Architecture Group 2018). The

CAG RFG is a TCL (Ousterhout et al. 2023) based code generator. Most importantly it generates hierarchical verilog HDL code besides an HTML documentation and a generic XML representation of the registerfile tree structure. The resulting registerfile offers a 64 bit address space and registers with a width of up to 64 bit. Registers are subdivided into individual fields and offer two types of access. *Software interfaces* provide address-based access to whole registers, while *hardware interfaces* offer granular direct access to the registers' fields for the attached hardware units. Independent access rights for both interface types, such as read- and / or write access can be specified to the register fields. It should be noted that the software interface's handshake protocol enforces atomicity, i.e. there can only be one pending access at a time. The resulting hardware tree is however hierarchically pipelined such that large distances can be bridged without exceeding the timing of single clock cycles. The generated HDL description is also hierarchically distributed throughout a module hierarchy, as provided by the user through the TCL registerfile definition. This allows the modular distribution of sub-registerfiles across the design hierarchy to those places where they functionally belong. Notably, the EXTOLL network offers direct access to the address space of this configuration and status registerfile (cf. Giese et al. 2012; Thommes 2018).

3.3 The Software Stack and Experiment Flow

A detailed overview on the existing current BSS-2 software stack with references for further reading can be found in (Müller, Arnold, et al. 2022). Figure 3.5 shows a block diagram of the software stack's layering architecture.

Generally, the BSS-2 software architecture is roughly divided into four levels. On the lowest level, the raw transport layer communication (*Host-ARQ*: Müller, Schilling, et al. 2018) and connection handling (*hxcomm*: Electronic Visions(s), Heidelberg University n.d.(g)) with the hardware system is implemented. Additionally, this level contains the Co-Simulation layer (*flange*: Electronic Visions(s), Heidelberg University n.d.(c), cf. Section 8.1.4).

The second level contains hardware abstraction libraries such as FPGA Instructions (*fisch*: Electronic Visions(s), Heidelberg University n.d.(b)), Coordinates which are the basis for hierarchical address calculations (*halco* (Electronic Visions(s), Heidelberg University n.d.[e])) and Containers abstracting the bit-structure of configuration and status registers in the system (*haldls*: Electronic Visions(s), Heidelberg University n.d.(f)). The Hardware Database (*hwdb*) contains basic information about the physical system like e.g. ip-addresses and node-ids assigned to individual systems. The Embedded Runtime library (*libnux*: Electronic Visions(s), Heidelberg University 2022) abstracts the hardware from the PPUs' point of view. Besides this, the *stadls* layer implements functions for executing playback programs that have been defined either in the realm of *fisch* or *haldls*. In any case, playback programs are compiled to raw FPGA instructions using *fisch* and transferred to the FPGA using a selected *hxcomm* connection instance via the respectively selected and available transport layer.

The third level divides between the two disjunct tasks of Experiment Description (*grenade*: Electronic Visions(s), Heidelberg University n.d.(d); Spilger 2021) and hardware Calibration (*calix*: Electronic Visions(s), Heidelberg University n.d.(a)). Thereby, the Experiment Description layer implements a conversion between a very abstract neuroscience perspective into a concrete representation onto the constraints and capabilities of the physical hardware system. The Calibration layer on

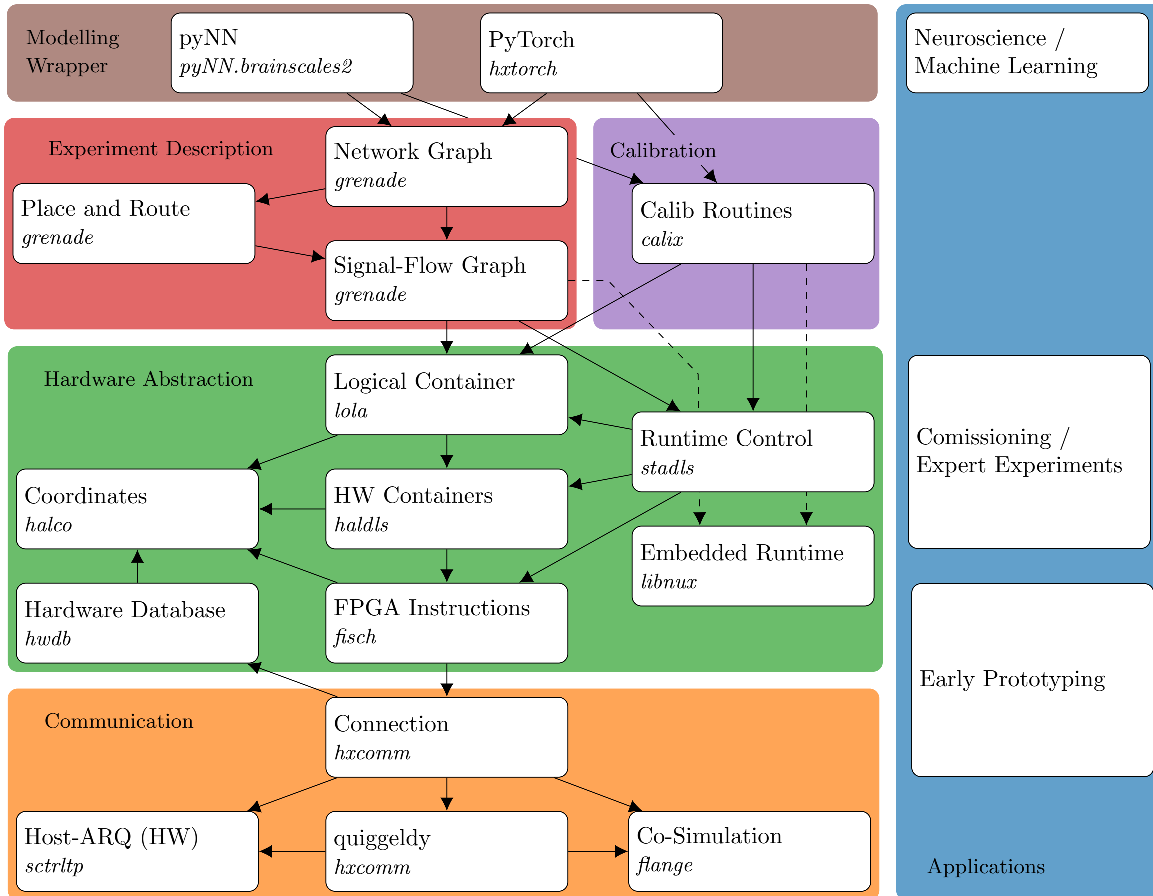


Figure 3.5: "Overview of the BSS-2 software architecture and its applications. **Left side:** Coloured boxes in the background represent the separation of the software into different concerns. White boxes represent individual software APIs or libraries with their specific repository names and dependencies. **Right side:** Various applications concerning different system aspects. The arrows represent dependencies in the stack, where the dependent points to its dependencies. For embedded operation additional dependencies on *libnux* are needed (dashed arrows)."
This Figure and caption text have been taken from (Müller, Arnold, et al. 2022).

the other hand aims to compensate for the effect of production variations between individual BSS-2 ASICs by applying analogue parameter calibrations.

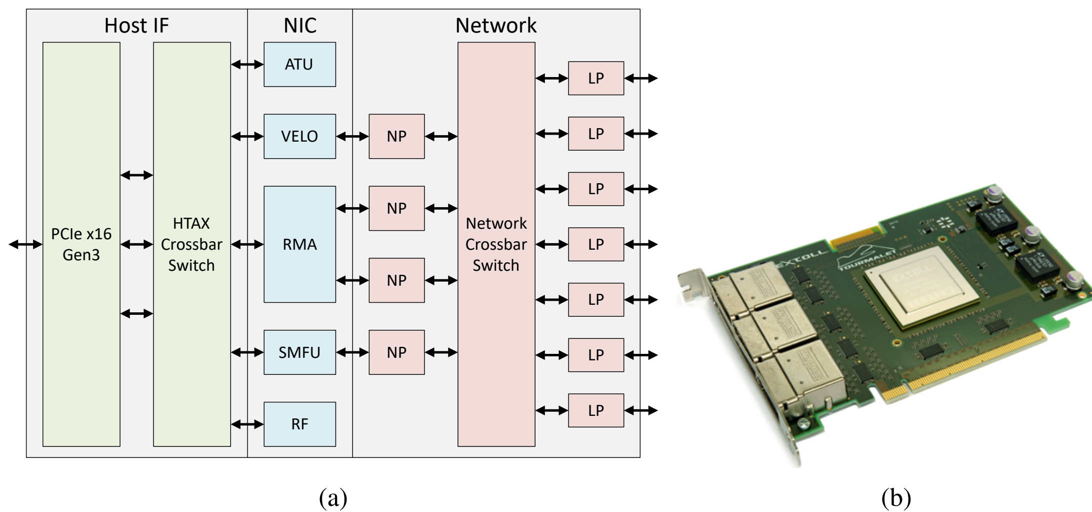
Last but not least, the topmost level implements wrapping layers for high level neuroscientific (*PyNN*, *pyNN.brainscales2*: Davison et al. 2009; Electronic Visions(s), Heidelberg University n.d.(j)) and machine learning (*PyTorch*, *hxtorch*: Electronic Visions(s), Heidelberg University n.d.(h); Paszke et al. 2019) experiments.

A general *PyNN* experiment is defined as a number of typed neuron populations that are connected through so-called projections. Roughly for BSS-2, populations can contain `HXNeurons`, offering control over the analogue parameters of BSS-2 hardware neurons, or so-called `SpikeSourceArrays`, defining an array of spike events to be inserted as stimulus input into the BSS-2 chip at specific times. Projections can implement different connection types like *All2All* or *One2One* and also define the synapse weights, including whether the connection shall be excitatory or inhibitory. Furthermore, projections are used to define which hardware neurons shall be stimu-

lated by the `SpikeSourceArray` populations. The individual neuron-instances in a population will thereby be mapped to spike labels that will be emitted by the respective hardware neuron or the input spike train from the FPGA. The *PyNN* experiment library operates on a global *simulator state*. An experiment therefore always begins by initialising the *simulator* by a call to `pynn.setup()`. After this, the populations and projections are created and automatically stored in the global *simulator state*. Having the neural network topology defined, the experiment is executed for a specified period of simulated time by calling `pynn.run()` and finally the *simulator state* is cleared by a call to `pynn.end()`.

Generally the experiment execution is partitioned into a pre-realtime section, one or multiple realtime sections and a post-realtime section. Thereby the realtime sections encompass the main neuromorphic experiment emulating the neural network on the BSS-2 hardware. Before that, the hardware is configured accordingly and afterwards the configuration is cleaned up into a reset-state where the next experiment can operate on the chip without noticing the previous configurations. Among other things, the `pynn.setup()` method provides arguments for injecting configuration playback programs at any point between these experiment phases, as well as the specification of a particular transport layer connection.

4 The EXTOLL Network Technology



(a) (b)

Figure 4.1: The EXTOLL Tourmalet network ASIC design and interface card.
(a) Block diagram of the EXTOLL networking hardware design; modified from (EXTOLL GmbH 2017a; B. U. Geib 2012). Packets are routed through the network by the *Network Crossbar Switch*. While the Link-Port (LP) interfaces the crossbar to the physical links, the Network-Port (NP) interfaces towards functional units implementing different communication protocols like RMA, VELO and distributed shared memory (SMFU). An Address-Translation-Unit (ATU) handles conversion between virtual and physical addresses, while the Register File (RF) provides an interface for configuration and status readout. The host computer is interfaced through an PCIe interface which is connected to the functional units via the HyperTransport Advanced Crossbar (HTAX).
(b) Picture of an EXTOLL Tourmalet PCB, taken from (EXTOLL GmbH 2017b).

The EXTOLL interconnection network has been developed by the EXTOLL company, based in Mannheim[2] which is a spin-off from the Computer Architecture Group (CAG) at the University of Heidelberg[3]. The design goals were to provide high bandwidth and message rates with low latencies for interconnecting high performance computing clusters. EXTOLL implements a direct, switch-less network, capable of connecting nodes in a 3D-Torus or any other topology with a node-degree of up to 7. A seventh besides six regular links is supplied for additional connections to special nodes. The global address-space supports up to $2^{16} = 65{,}536$ nodes. Multicast communication is supported by hardware with up to 64 multicast groups. A built-in barrier and interrupt mechanism supports global interrupts across the network with very low skew of only a few clock cycles. This interrupt mechanism will play an important role in this thesis in synchronising systime counters across several BSS-2 systems (cf. Section 7.2). The accuracy of the interrupt operation synchrony will be evaluated

[2] www.extoll.de/
[3] www.ziti.uni-heidelberg.de/ziti/en/institute/research/computer-architecture-group

in Section 8.6.3.

Originally, the EXTOLL Tourmalet ASIC was announced to provide a link-bandwidth of up to 120 $\frac{\text{Gbit}}{\text{s}}$ and a tiny hop-latency, i.e. the latency of one switching hop through the network, of down to 60 ns, as well as messaging latencies below 600 ns and sustained message rates of more than 100 million messages per second, running at a clock frequency of 750 MHz. However, due to technical reasons, the core frequency could only be realised to 630 MHz and especially the Tourmalet boards used for this thesis run at a frequency of 600 MHz. This frequency reduction leads to slightly reduced core performance numbers with a resulting link bandwidth of 96 $\frac{\text{Gbit}}{\text{s}}$ and a hop-latency of 75 ns. More details on the frequency scaling and other adaptions of the network to the BSS-2 system will be elaborated in Section 8.2.2. The core performance characteristics and design principles of the EXTOLL network have been scientifically evaluated and published in (Fröning, Nüssle, et al. 2013; H. Litz et al. 2008; Nüssle, B. Geib, et al. 2009; Nüssle, Scherer, et al. 2009). Especially the high sustained message rate and low message latencies make the EXTOLL network suitable for accelerated neuromorphic computing.

Figure 4.1a shows an overview, containing the major building blocks of an EXTOLL network ASIC while Figure 4.1b shows a picture of the Tourmalet network card on which the EXTOLL chip is mounted and can be inserted into a standard computer via a PCIe slot. Notably, the EXTOLL network chip already contains everything necessary to build a full working interconnection network. Especially the packet switching hardware is included internally, forming a switch-less direct network (cf. Section 2.3.2.2). The following sections will give an overview of the main building blocks, summarising their functionality. More details and references can be found in the work of (B. U. Geib 2012).

4.1 The Network Partition

The network partition, shown in red on the right side of Figure 4.1a contains everything that is necessary to form a network of nodes. It consists of multiple Link-Ports (LPs), interconnected to multiple Network-Ports (NPs) through a crossbar switch.

4.1.1 The Link-Port

The Link-Ports (LPs) (developed by Burkhardt 2007, compare Section 4.1.7 of B. U. Geib 2012), corresponding to the OSI Link Layer (L.2), guarantee the error free and efficient transmission of packets through the network. Possibly erroneous packets are detected through a strong Cyclic Redundancy Checksum (CRC) code that is attached to each packet before sending it over the physical link. When an error is detected while receiving a packet, a retransmission is requested from the sending side. The latency introduced by retransmission of faulty packets is minimised by implementing the error detection and retransmission on the lowest possible level in the network, as opposed to an end-to-end retransmission which would cause a full round-trip latency.

Each *link* is divided into 12 *lanes* that are combined to groups of four *lanes*, so-called *quads* that can be separately activated or deactivated. Each lane is encoded using an 8b-10b code in order to protect the physical transaction words against transmission bit-errors (cf. J. Schmitt 2017).

4.1.2 The Barrier Unit

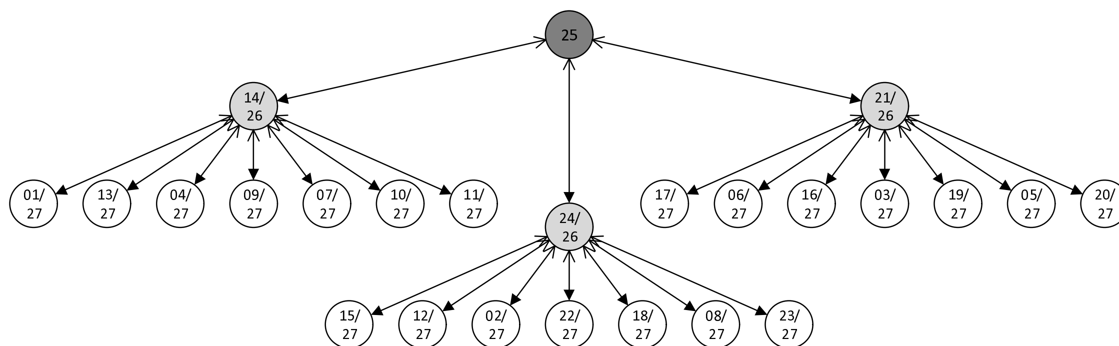
The EXTOLL Barrier unit provides efficient hardware support for global barrier- and interrupt operations in the network (developed by Burkhardt 2007, compare Section 3.4 of Burkhardt 2012). The EXTOLL card provides hardware units for up to 16 barrier- and 4 interrupt operations in parallel. The Barrier units are directly attached to the EXTOLL Link-Ports in order to avoid the additional latency through the network crossbar. Barrier messages are also forwarded with higher priority than normal network traffic to avoid congestion for the Barrier- and Interrupt operations.
The *Barrier* operation provides global synchronisation in a way that a set of N participating nodes and processes wait until they have all reached a certain point of operation before continuing. This does notably not provide an exact point in time where all the different units will continue, but rather a lower bound in time t_{cont} for continuing the operation:

$$t_{\text{cont}} = t^{i}_{\text{barr, received}} > t^{j}_{\text{barr, reached}} \quad \forall i \in N,\ \forall j \in N \tag{4.1}$$

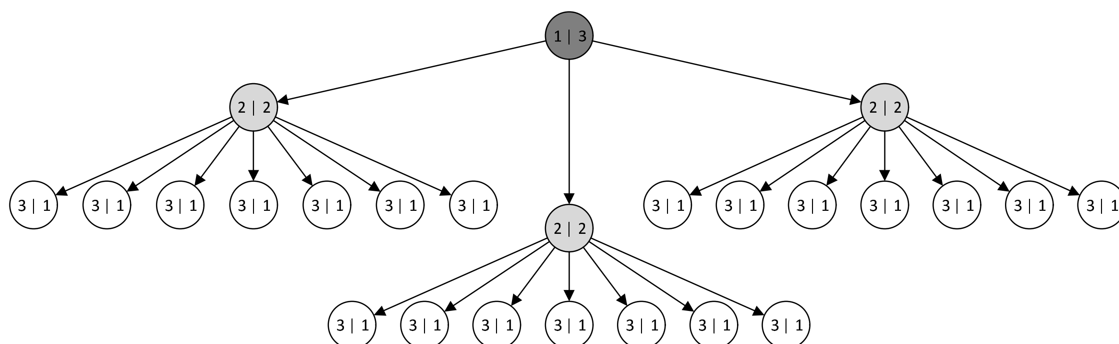
The *Global Interrupt* on the other hand, synchronises the participating nodes globally to a common point in time. This can for example be used to synchronise a globally distributed timestamp counter to the same value.

$$t_{\text{cont}} = t^{i}_{\text{int, received}} = t^{j}_{\text{int, received}} \quad \forall i \in N,\ \forall j \in N \tag{4.2}$$

Both synchronisation paradigms make use of a virtual tree structure, imposed into the physical network by configuration. Figure 4.2 schematically shows a Barrier- and an Interrupt tree.
In a *Barrier operation* (Figure 4.2a), each process notifies its hardware unit that it has reached the barrier. A leaf node's hardware unit will then send an *up-message* to its parent in the network tree, when all the subscribed processes have reached the barrier. Similarly, the parent node will forward an *up-message* when all its own processes and child nodes have reached the barrier. When the *up-messages* have reached the root node, a *down-message* will be propagated along the tree. Each node receiving a down message will immediately release its subscribed processes from the barrier. This release of the nodes from the barrier is however, not synchronous because the *down-messages* propagate through the levels of the tree and release the nodes on their way immediately. Also it does not account for latency differences on the individual physical links between the nodes.
For the *Interrupt operation* (Figure 4.2b), only the *down-phase* of the *Barrier operation* is used. The operation is directly triggered by a master process at the root-node. In order for the *Interrupt operation* to yield a common point in time at every node, the messages must have a constant latency at each hop through the network. Although Interrupt- and Barrier messages have higher priority than normal network traffic, they might still have to wait for another packet that is already underway through the link. In order to provide constant latency, Barrier-messages are inserted onto the link with highest priority, only after waiting for the longest time, a normal network packet could possibly take for transmission across the link. Thereby, any packet that is currently using the link will be finished before the Interrupt message is inserted. Variations in latency between the individual links and the position in the interrupt tree are compensated by a programmable wait-timer in each unit. After all links have been measured using a special measurement mechanism built into the interrupt units, these delay counters are programmed such that each interrupt unit will release the interrupt notification globally at the same point in time, depending on the result of the round-trip latency



(a) The Barrier tree operation. An exemplary order of visits is indicated by the numbers written to the node circles. Each node is visited two times.



(b) The Interrupt tree operation. The first number written to the node circles gives the visiting order of the interrupt messages, the second number gives an exemplary wait-count, compensating the different arrival times of the interrupt message at each node.

Figure 4.2: The Barrier- and Interrupt tree operations. *Up-messages* are displayed using open arrow-heads while *down-messages* are visualised using filled arrow-heads.

measurement and the nodes position in the interrupt tree.

The EXTOLL Barrier Unit will be used for globally synchronising the systime counters of the BrainScaleS-2 FPGAs in the later course of this thesis (cf. Section 7.2.2).

4.1.3 The Network Crossbar

The network crossbar (cf. Section 3.8 of B. U. Geib 2012) is the central component of the EXTOLL network, as it performs the basic routing operation of packets through the network and towards their final destination unit. Thereby it basically implements the OSI network layer (L.3) together with the NPs (cf. Section 4.1.4).

The crossbar can execute arbitrary routing algorithms by implementing a table-based routing mechanism (cf. Section 2.3.4, compare Section 3.8.1 of B. U. Geib 2012). Thereby, nodes are labelled with a unique 16 bit node-id throughout the network. Packets are addressed using the unique node-id of their destination node. When a packet reaches the crossbar at an input port, the destination node-id is used as an index to the routing table, yielding the output port to which to forward the packet.

In order to reduce the required memory space while maintaining the same lookup latency and address space size, the EXTOLL crossbar implements hierarchical routing tables. Nodes are organised in 64 network segments using the upper 6 bit of the node-id. Each segment can contain up to 1024 nodes,

distinguished by the lower 10 bit of the node-id. The routing decision is made by two simultaneous table lookups. If the global part of the destination address does not match the local node-id, the global routing entry is used for forwarding the packet. Otherwise the packet has already reached its destination segment and is now forwarded using the local routing entry. If the full address matches the local node-id, the packet has reached its destination and will finally be forwarded to the internal unit, requested through a field in the packets header (cf. Appendix B.2).
Deadlocks can be avoided by applied routing algorithms through the support of 2 Virtual Channels (VCs) (cf. Section 2.3.3.5). Adaptive routing is also optionally supported by the use of a dedicated adaptive Virtual Channel. The routing can make use of 4 so-called Traffic Classes, each providing a different route through the network.
The EXTOLL crossbar uses the Virtual Cut-Through Switching (VCT) strategy (cf. Section 2.3.3.3).
The input buffers feature Virtual Output Queues (cf. Section 2.3.3.5) in order to avoid Head of Line Blocking (cf. Section 2.3.3.4).
Multicast operation is also supported by the crossbar hardware through the use of a dedicated multicast routing table. For this purpose, up to 64 multicast groups can be defined. A packet which is addressed to a multicast group is then replicated sequentially in the crossbar towards all output ports yielded from the multicast routing lookup.

4.1.4 The Network-Port

The Network-Ports (cf. Section 4.5 of B. U. Geib 2012) establish a connection between the Functional Units (FUs) in the Network Interface Controller (NIC) partition and the network crossbar. A credit-based flow control mechanism (cf. Section 2.3.3.2, compare Section 3.8.6 of B. U. Geib 2012) is implemented here, as the NPs form the endpoints of a connection across the EXTOLL network. The correct framing of packets is also ensured here at the entry-point into the network.
The NP is divided into two main subunits, the NP *sender* and the NP *receiver*. The *sender* is responsible for the transmission of packets towards the crossbar and into the network, while the *receiver* forwards incoming packets to its respective FU. In particular, the *sender* appends an End Of Packet (EOP) cell to each packet and checks that it has enough credits to its disposal in order to send the packet back-to-back according to the EXTOLL specification. A small buffer is instantiated here to store packets until enough credits are available.
The *receiver* on the other hand needs to instantiate a larger buffer, so to be able to take as much data as it gave out credits to sending sites before. When the attached FU has read the data from the buffer, credits are immediately returned to the sending side. For error handling, the NP offers two different modes. In VCT mode, packets are partly forwarded to the FU as they are received. If the functional unit however, is not able to handle signalled packet errors, the NP can be operated in a SAF mode. In this mode, incoming packets are buffered, until they can be determined to be error-free. If an error is detected, the buffered packet will be invalidated and dropped. However, the data will not be lost though, as the LP will already have requested a retransmission.
As the NPs use the same uniform crossbar port interfaces as the LPs, these two units are compatible for direct connection. This direct compatibility is made use of in the BSS-2 FPGA, where only one link and target functional unit is present (cf. Section 3.2.4). In this case no network crossbar is needed and the NP and LP units can be directly connected.

4.2 The NIC partition

The Network Interface Controller (NIC) partition collects all the functional units implementing low level protocols for communication between host computers throughout the network. The direct implementation of these protocols in hardware plays an important role towards the low end-to-end communication latencies achieved through the extoll network. With the help of these hardware units a great part of the communication that would otherwise be handled by the operating system can be offloaded from the CPU to the hardware. This greatly reduces the communication overhead that is usually experienced in high performance computing systems.

4.2.1 The RMA Unit

The Remote Memory Access (RMA) Unit (Nüssle, Scherer, et al. 2009, compare Section 7.5 of Nüßle 2008) is implemented to efficiently support atomic remote DMA transfers of data chunks. For this purpose it provides an instruction set consisting of *put* and *get* operations as well as special *notification* and *atomic lock* operations.

The *put* operations transfer a sized chunk of data from a specified local DMA address to a specified remote memory address. Automatic notifications to the requesting process can be triggered upon completion of the operation at the requesting side as well as at the completing side. Additionally, there is an *immediate put* operation that transports a small amount of data directly with the command itself, skipping the DMA access at the requesting side. Similarly, the *get* operations request a remote node to write a sized chunk of data from its remote memory to the local memory of the requesting node. Again automatic notifications can be triggered upon completion at the requesting, responding or completing site. A special *notification put* instruction can be used to directly send a small amount of data to the notification queue of a remote process. This can be useful for the implementation of various communication protocols. The *atomic lock* operation performs an atomic fetch-compare-and-add (FCAA) operation at a remote location (cf. Section 7.5.4 fo Nüßle 2008). This operation basically compares the value of the remote location with a given value. Upon success, i.e. the value at the memory location is less or equal to the comparison operand, the operation adds the second operand to the retrieved value, writes back the result to the memory location. Finally, the requesting side is notified about the result of the comparison. This special instruction can be very useful for the efficient implementation of distributed parallel algorithms.

The microarchitecture of the Remote Memory Access (RMA) unit is divided into three sub units, the *Requester*, *Responder* and *Completer* (cf. Section 7.5.4 of Nüßle 2008). The implementation of the different parts of the operations described above is distributed across these units. The *Requester* is responsible for sending request packets to the remote node with data it retrieved from the local memory. The *Completer* on the other hand is responsible for receiving and interpreting the response packets from the remote node. Finally it will write the result of a *get* instruction or the content of a *put* command to the local memory or insert received notifications to the requesting process's notification queue. In between, these two units, the *Responder* will fetch the requested data for *get* commands from the memory and send the response packet back to the requesting node. As the *Responder* is responsible for receiving requests as well as sending responses, it makes use of a whole NP interface, while *Requester* and *Completer* can share one NP interface, as they implement

complementary responsibilities in sending and reception of packets.
Memory addresses in the transmitted commands can either be interpreted as physical addresses, directly used to access the main memory or as virtual addresses, which are generally better suited for user level software. In case of virtual addresses, the RMA unit makes use of the Address-Translation-Unit (ATU), also implemented in the EXTOLL Network Interface Controller (NIC) (cf. Chapter 5 of Nüßle 2008). This unit will then translate the provided virtual address to a physical one that can then be used for DMA access to the local memory.
In the scope of (Thommes 2018) and this thesis, the RMA unit is used for communication of the BrainScaleS FPGA with the controlling experiment host. For this purpose, a simplified RMA microarchitecture (called NHTL) was developed in (Thommes 2018), only supporting a basic subset of the whole RMA instruction set (*put*, *get responses* and *notifications*). In the scope of this thesis, the subset was extended by *get* requests for Remote Registerfile Access (RRA) master operation (cf. Section 7.4 and Section 7.6.4). In order to further simplify the microarchitecture and make do with a single NP as compared to two NPs in the original design, the receiving functionality of the *Responder* was merged with the *NHTL Completer* which will now forward *get* requests to the *NHTL Responder*. Similarly, the functionality of the *Requester* was merged into the *NHTL Responder*. This approach is similar to the one used for the simplified RMA microarchitecture in (J. Schmitt 2017).
A reference of the RMA packet types and their headers in the scope of this thesis can be found in Appendix B.2.

4.2.2 The VELO Unit

The Virtualised Engine for Low Overhead (VELO) Unit, as described in (H. Litz et al. 2008, Section 4.6 of B. U. Geib 2012 and Section 7.4 of Nüßle 2008), has been developed to support small messages with a minimal latency and overhead between software processes of different network nodes. This is achieved by writing messages directly to the device address space from user software at the sending side and polling a ringbuffer at the receiving side which is filled automatically with arriving messages through DMA accesses by the hardware. On the sending side, this approach avoids the latency introduced by a DMA access. On the receiving side, the polling strategy avoids the latency of kernel-interrupt context switches that would occur with directly sending messages to the receiving process. It also gets rid of the necessity to transmit remote write addresses, as those are automatically determined by a ringbuffer controller in the receiving hardware. This Virtual Ringbuffer Handler (VRHD) (Section 2.4 of B. U. Geib 2012) is designed to handle the mapping of a segmented physical address space to a multitude of continuous virtual address space ringbuffers. This same VRHD unit is also used to write RMA notifications to the host.
The VELO unit is divided into two subunits, the *VELO Requester* and *Completer*. The *Requester* collects message parts from the host interface and combines them to a single network packet. While the source process id, as well as the traffic class (cf. Section 4.1.3) are provided by the device address to which the message is written, the destination information like the target node- and process id, as well as the message's length are conveyed with a status word, resembling a minimal packet header. The *VELO Completer* on the other hand receives packets from the network and, after performing some integrity and security tests, forwards them towards the host interface where they are written to main memory. The memory target address is thereby provided by the VRHD.

In order to avoid data loss, a flow-control mechanism has to be implemented in software on top of the VELO unit (Prades et al. 2012). The flow-control is implemented as a protocol software library layer as a dynamic credit-based mechanism. The dynamic feature of this flow-control mechanism is to split the receiving buffer into dynamically sized virtual regions and assigning those to the sending remote processes. The size of the respective buffer-regions correlates to the frequency of traffic, the respective remote process is sending.
As this software-based flow-control is essential to the correct function of the VELO unit and reimplementing this exact algorithm in the BrainScaleS FPGA would have been a complex task, it was decided to rather use the RMA unit for communication of the BrainScaleS system FPGA with the host computers. Actually, the communication mechanism developed in (Thommes 2018) resembles a simplified version of VELO communication, implemented using RMA packets (cf. Section 7.4 and Appendix B.2). One disadvantage of this choice is however that it introduces some additional latency at the sending side through not making use of the direct address-space transmission scheme of VELO. However, this latency can well be hidden, as the data transmitted from the host to the BrainScaleS FPGA consists of precompiled playback programs, usually forming large chunks of data that are there realtime-executed towards the actual neuromorphic hardware (cf. Section 3.2.2). Indeed, (B. U. Geib 2012) and (H. Litz et al. 2008) stat that the latency gap between RMA and VELO notably shrinks with increasing message size.

4.2.3 The SMFU Unit

The third functional unit in the EXTOLL NIC is the so-called Shared Memory Functional Unit (SMFU) (Fröning and H. Litz 2010). As the name of the unit already tells, it provides a global shared memory address space throughout the network. This is achieved by directly tunnelling load and store instructions from the host interface bus (e.g. PCIe) into the remote node. This offers transparent remote memory access to software, in the same way, it would use local memory. This is especially useful for very fine grained access patterns like e.g. process synchronisation.

4.2.4 The Registerfile

The registerfile implements configuration and status registers across the whole design. These registers are made available through a software interface, as well as a hardware interface. The relevant code for the HDL description, Linux kernel driver, HTML manual and functional verification is generated from a common source by a custom tool-chain. This tool-chain has been under ongoing development together with the EXTOLL hardware itself and therefore changed in the internal details over the years (Computer Architecture Group 2018; B. U. Geib 2012; Nüßle 2008; Wenzel 2018). The version used in the scope of this thesis is (Computer Architecture Group 2018), which is briefly described in Section 3.2.5.2.

4.3 The Host Interface

The EXTOLL design was originally developed offering a HyperTransport (HT) interface. HT is an open specification for a CPU interface bus mainly implemented in AMD processors. Because of its

direct connection to the CPU, HT offered a very low latency (Slogsnat et al. 2008). In order to provide a more standard interface, also a PCIe interface was developed and integrated into the EXTOLL chip. The design offers static switching between these two interfaces, meaning that one can either use one or the other, but dynamical switching during operation is not supported (Section 4.1.1 of B. U. Geib 2012). The EXTOLL Tourmalet network card now offers only the PCIe interface, as the HT bus is not widely supported anymore in recent systems.
In order to connect the functional units, described in Section 4.2 to the single host interface selected for operational use, an on-chip network crossbar called HyperTransport Advanced Crossbar (HTAX) was developed and described in (H. H. Litz 2011). It offers a protocol-agnostic and highly configurable network-on-chip design. As the original EXTOLL design was fully developed towards the HT interface, all functional units operate across the HTAX on the HToC protocol, which was especially developed for this purpose.

4.4 The Software Stack

The driver- and API software stack supporting the EXTOLL network is shown in Figure 4.3.

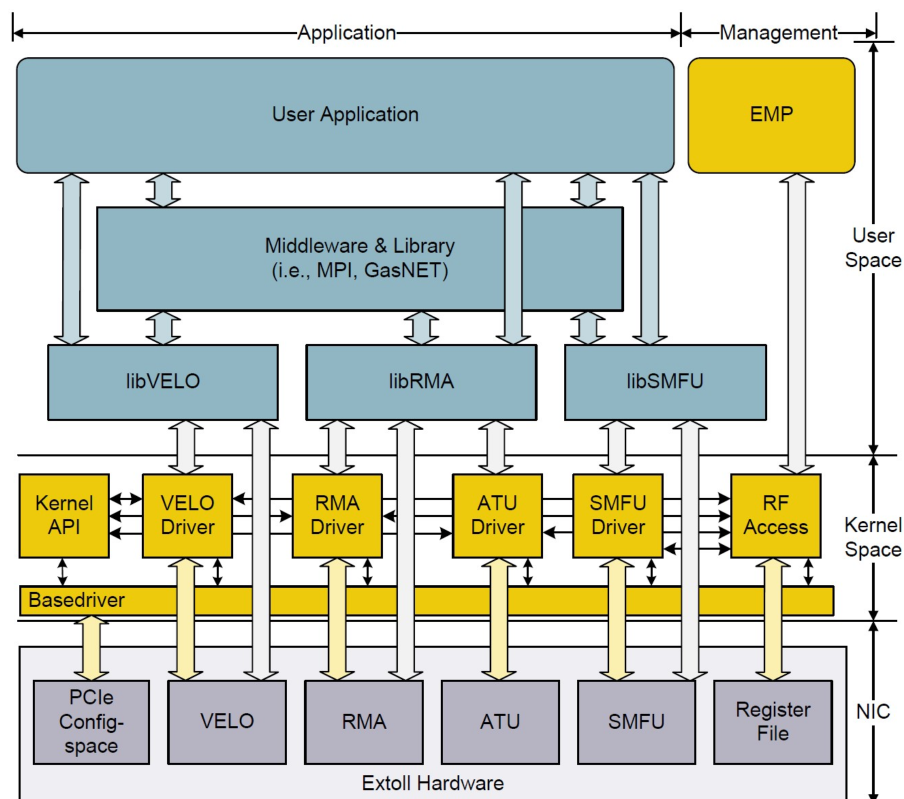


Figure 4.3: The EXTOLL software stack. Communication with the hardware is managed by kernel space drivers, while user space applications are provided with API libraries and transparent middleware like e.g. Message Passing Interface (MPI) and GASNet (Fröning 2015). This Figure is taken from (N. A. Buwen 2019)

On the lowest layer, multiple linux kernel drivers handle the physical communication with the respective hardware units, described in Section 4.2, via the host computer's PCIe bus system. On top of

the kernel drivers, user space APIs provide access to the functionality of the RMA (cf. Section 4.2.1), VELO (cf. Section 4.2.2) and SMFU (cf. Section 4.2.3) hardware units. So-called middleware libraries like e.g. MPI and GASNet (Fröning 2015) may be specifically implemented directly on top of the EXTOLL API layers in order to transparently provide network access to existing user application software.

The Extoll Management Program (EMP) has been developed by Tobias Groschup at the EXTOLL company and is provided for bringing up a physical network after adding nodes or changing the topology. This tool mainly performs three steps to configure the Tourmalet nodes and start network operation. First, it will do a discovery run to find the physically connected network topology like for example a grid or torus topology (cf. Figure 2.6). This is done, by reading the status Registerfiles of the all connected LPs and retrieving the Global Unique Identifiers (GUIDs) and building an internal representation of the found physical topology. For this task, the tool assigns temporary routes and Node IDs in a depth-first-search. Second, the tool will calculate a possible routing configuration or check, whether a given routing configuration from a source file will apply to the physical topology, found in the first step. Finally, the third step is to write the calculated routing configuration to the Registerfiles of all the crossbar link ports, i.e. to write the routing tables on the involved nodes. The EMP tool also enumerates the configured nodes to the driver and can be used to print out status information about the network and the contained nodes.

Part II

Event Communication

5 Event Communication Principles and Systems

As motivated in Chapter 1, large scale neuromorphic computing systems are special purpose supercomputers, based on novel computing paradigms which lean on the knowledge obtained about the computing principles of nervous systems in biology. As such, they exhibit very special requirements to Quality of Service which shall be derived and motivated from literature in Sections 5.1 to 5.3. Following this introduction, Section 5.4 will briefly describe methods to obtain these requirements for Quality of Service (QoS) for neuromorphic spike event communication.
Besides that, this Chapter also gives a short overview on some existing large scale neuromorphic systems and their respective approach to neural event communication (cf. Section 5.5). Finally unique requirements of neuromorphic event communication in packet-based interconnection networks are explained while giving a first overview on the event communication architecture, developed and implemented for the BSS-2 system in this thesis (cf. Section 5.6).

5.1 Event Communication in General

Event communication is an important paradigm in modern large scale surveillance and control systems. Real world applications like e.g. weather or water level observation and forecast, operation of power grids, driven by the real time electrical demand, or home automation systems represent systems that gather measurement data in order to perform overall evaluating computations. Collecting high resolution data streams from a high number of measurement sites would however impose high demands on communication bandwidth and computation frequency for evaluation at a central site. Therefore the evaluation is spread across the whole system to compress the data streams and only important signal changes are communicated as events towards a central evaluation site. This paradigm is also referred to as *ubiquitous computing* (Rodrigues et al. 2010; Albrecht Schmidt 2002).
Another application of event communication for compression of data streams are event-based or neuromorphic cameras (Li et al. 2017; Lichtsteiner et al. 2008; Liu et al. 2017). These cameras take inspiration from biological vision systems. Instead of streaming a brightness trace for every pixel in sequential readout order, the pixels in these cameras independently and asynchronously report only changes in their respective brightness. This has the great advantage of directly depicting movements as opposed to a sequence of static images.
In addition to event based sensing, biological systems also perform event based computation, as described before in Section 2.1. Neuromorphic Computing (Section 2.2.1.3), aiming to imitate the biological event based computation, takes event communication to a whole new level. When scaling neuromorphic computing systems to implement large neural network models, spike events have to be communicated between the different parts of that system.

As the goal of this thesis is to propose and implement an event communication mechanism for the BrainScaleS-2 neuromorphic computing system (cf. Chapter 3), the next Section will focus on the particular aspects of event communication in neuromorphic computing systems in the form of Spiking Neural Networks (SNNs).

5.2 Event Communication for SNNs

In SNNs, as described in Section 2.2.1.3, neurons exchange information by emitting spike events to the synapses of connected neurons. The presynaptic neurons emit such spike events when their membrane potential exceeds a threshold value. The postsynaptic neurons, receiving these events react to them by changing their membrane potential according to the received synaptic input. Electronic emulations of SNNs have to detect the presynaptic spikes and communicate them to the receiving synapses.
In biology, each synaptic connection is directly and physically routed from the axon of a presynaptic neuron to the dendrite of a postsynaptic neuron. This vast connectivity is possible as biological neuronal networks grow as a three-dimensional tissue, dynamically evolving over time and using free intercellular space for real three-dimensional routing. With the current state of the art for VLSI Integrated Circuits (ICs) however, technical implementations of connection networks are constrained to a two-dimensional space that is at most extended by a few layers, stacked vertically above each other (Deng et al. 2005; J. Kim et al. 2020; Najmaei et al. 2022; Takawadekar et al. 2021). Importantly, inter-chip networks are even more constrained as every synaptic connection, crossing the chip boundary, has to be mapped to a pin or pad either on the chip's one-dimensional border (in case of wire-bonding) or on the two-dimensional area of its top layer (in case of flip-chip bonding or wafer-scale integration). Therefore physical connections in a technical implementation are strictly limited in their number. However, the bandwidth of serial data transmission across these connections can be much higher than required for the realisation of a single neural connection. It is therefore feasible to bundle and compress the information stream of multiple synaptic connections and transport them between the technical neuron instances using a shared communication infrastructure.

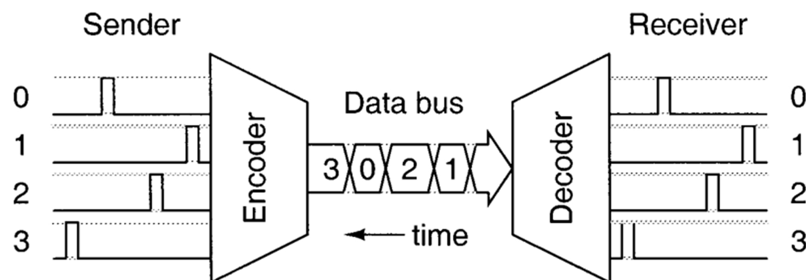


Figure 5.1: Schematic of the Address Event Representation coding; taken from (Culurciello et al. 2003).

As stated in Section 2.1.1, the temporal shape of action potentials (aka. spike events) does not carry significant information. Instead information is conveyed through the spatial and temporal distribution of the spikes. The Address Event Representation (AER) (Culurciello et al. 2003; Mahowald 1992; Sivilotti 1991) encoding therefore represents spike events by a unique label corresponding to

the address of the emitting source neuron. If the interconnect latency is not constant and has variations, comparable in magnitude to the time-intervals between subsequent events, a timestamp has to be added to compensate for this jitter at the destination (cf. Section 5.3.2 and Section 5.4). This timestamp will then represent either the time when the source neuron has fired or the time when the event shall arrive at the target synapse. This label- and timestamp representation can then be communicated across the shared interconnect infrastructure. Figure 5.1 schematically depicts this principle of AER coding without timestamps. The interconnect is hereby approximated as a simple data bus (cf. Figure 2.5) and spike times are encoded in the signal edges.

5.3 Quality of Service Requirements for Spike Communication

In general, the term Quality of Service (QoS) refers to required and offered characteristics of transmission across an interconnect for a specified type of transmitted data. This includes *reliability* with respect to correctness and completeness of transmitted data, *bandwidth*, as well as bounds on the transmission delay (*latency*) and its variation (*jitter*).

The following subsections will respectively discuss the requirements, imposed on a technical implementation of spike event communication between neurons.

5.3.1 Spike Latency

The axonal delay is defined as the time, an action potential takes to travel from a source neuron's soma to the receiving neurons' synapses. In (Swadlow et al. 2012) an extensive review is presented on biological findings about the axonal conduction delays. The reviewed empirical studies report a large biological variety in the spike latency between different neural populations as well as inside populations. Other studies show that exactly this spatial variation of delays has a large effect on the dynamics of Spiking Neural Networks (Brunel 2000; Hornung 2020). In biology, these delays largely depend on the functional purpose of the connecting neurons in the respective nervous system as well as the species, age, health and environmental circumstances of the individual organism under investigation (Swadlow et al. 2012).

The delay itself depends on the distance in terms of physical connection length between the neurons. Besides that, the transmission speed of the action potentials also depends on the physiology of the respective axon. Basically, the conduction velocity of the axons rises with increasing axon diameter. Besides this basic effect, axons can also be surrounded by a myelin sheath that acts as an electrical insulator with high resistance and low capacitance. This largely increases the conduction speed across myelinated axons relative to non-myelinated ones with the same diameter. In biology, myelination is harnessed mainly for thick axons above diameters of approximately 0.3 µm (Waxman et al. 1976) to further improve their conduction speed over longe range distances.

(Swadlow et al. 2012) also reports numbers for the axonal delays from many different empirical studies. The delay is reported to be widely varying across different neural connections in different species. Short range connections internal to a rats cortical microcircuit (Potjans et al. 2012) are reported to be around 1.1 ms while long range connections between different brain regions are mostly reported to be in a range between 1 ms and 50 ms. Of course this range is not exclusive, so there are

also connections having delays below 1 ms or even up to 130 ms. For reference compare Table 1 in (Swadlow et al. 2012).

With the BSS-2 acceleration factor of 10^3 and its continuous time operation (cf. Chapter 3), these numbers lead to a hardware latency requirement in a range of 1 µs to 100 µs.

This large range of biologically observed transmission delays is, according to the review of (Swadlow et al. 2012), generally believed to serve the purpose of synchronising presynaptic activity from different locations in the nervous system onto different postsynaptic receptors, leading to different postsynaptic reactions depending on the precise timing of distributed presynaptic activity. Additionally, in recurrently connected networks, the delay of the recurrent connections limits the timescale, on which incoming spike signals can be functionally correlated with the recurrent spike signals.

5.3.2 Spike Jitter

Another important aspect for neural information processing is the temporal variation (*jitter*) of axonal delays. As described in Section 2.1.3, one aspect of learning in biology is based on the temporal correlation of pre- and postsynaptic spike times, which is also referred to as STDP. Thereby, connections are potentiated in case of causal correlation and depressed in case of acausal correlation. The magnitude of potentiation or depression is scaled by an exponential decay. The time constants of this decay typically reside in the range of the neuronal membrane time constant (Abbott and Nelson 2000). The technical implementation on the HICANN-X supports STDP time constants in a range of 11.36 µs to 336.6 µs for causal correlations and 12.31 µs to 228.73 µs for acausal correlations (Friedmann et al. 2017). Membrane time constants are calibratable in a range of 0.5 µs to 100 µs and synapse time constants in a range of approximately 0.5 µs to 11 µs according to (Leibfried 2021).

Together, these numbers give an approximate constraint to the tolerable jitter of the spike latency. While the lowest possible STDP time constants of slightly above 10 µs correspond to 2500 4 ns clock periods, the lowest membrane- and synapse time constants correspond to 125 4 ns clock cycles. So a latency variation in the order of a few hundreds of clock cycles could generally be estimated acceptable from a model point of view with regard to the STDP time constant and some tens of clock cycles with regard to membrane and synapse time constants. It can be summarised that the jitter should be as low as possible in terms of clock-cycles and that with larger membrane-, synaptic- and STDP time constants also a higher spike jitter can be tolerated.

The spike event communication implementation, described later in Chapter 7, together with the existing spike event communication across the serial link between BSS-2 ASIC and FPGA (cf. Section 3.1.2 on page 37 and Section 3.2.1) is constructed to ideally guarantee jitter to be eliminated to a few clock cycles. This claim can however only be so precise as the precision of the synchronisation of the system time counters on FPGAs and ASICs. For the event transport between the FPGA and ASIC, (Rettig 2019a) claims negligible precision influence with respect to the neuron time constants. For the external spike communication, the precision of systime synchronisation is measured in this thesis (cf. Equation (8.11) in Section 8.6.3) to be in the order of $\pm 5\,\mathrm{clk}$ in the FPGA domain ($8\,\frac{\mathrm{ns}}{\mathrm{clk}}$), corresponding to $\pm 10\,\mathrm{clk}$ in the ASIC domain ($4\,\frac{\mathrm{ns}}{\mathrm{clk}}$).

5.3.3 Transport Bandwidth

As spike events are technically transported using AER coding, their size in terms of transmitted information is constant and depends on the size of the neuron address space and the size of the transmitted timestamps. The overall momentary data rate then depends on the number of neurons and the individual rates at which they fire. Ideally, the interconnect network should offer enough bandwidth to transport any rate of spike events that might be generated by the neuromorphic source device. Practically however, this might not be possible as any physical network implementation will have an upper bandwidth limit.

An exemplary estimation might assume a mean biological firing rate of 2.45 Hz to 33.9 Hz, reported by (Baddeley et al. 1997). The maximum biologically plausible spike rate can be estimated by the absolute refractory period, which is reviewed by (Boërio et al. 2004) to be in a range of approximately 0.6 ms to 4.5 ms. This minimum refractory period results in a maximum spike frequency of 1.67 kHz. With a speed-up factor of 10^3 and 512 neurons on the HICANN-X neuromorphic chip, this would result in a total average event rate of 853 MHz. On BSS-2, with a 14 bit spike label (cf. Chapter 3), a 15 bit external timestamp (cf. Section 7.1.3) and ideal encoding of the events onto the physical bitstream of the interconnect, this gives a required bandwidth estimation of approximately 24.7 $\frac{\text{Gbit}}{\text{s}}$. This maximum required bandwidth arises from all neurons on a chip hypothetically firing at a maximum rate of 1.67 kHz. However, one must also take into account neurons firing at significantly higher rates in a bursting or high activity regime, leading to a higher bandwidth requirement for short periods of time. Still, it is hardly realistic that all neurons on the chip are in a bursting state at once, so the previous estimation for maximal regular spiking is still good. Furthermore, it also a good assumption that not all neurons are connected off-chip and therefore a high amount of the total activity stays internal to the chip, not contributing to the required transport bandwidth.

A technical upper bound on the required interconnect bandwidth for external spike event communication can be obtained from the HICANN-X specification (cf. Chapter 3). According to (Johannes Schemmel, Billaudelle, et al. 2022), the L1 links in the HICANN-X can provide one event at every 250 MHz clock cycle. Four of these links are connected to eight high speed serial links towards the outside world. Due to serialisation overhead, and the limited serial bandwidth of 8 $\frac{\text{Gbit}}{\text{s}}$, (Karasenko 2020) reports a maximum event throughput of 250 MHz between the BSS-2 ASIC and an FPGA. With the above mentioned external event size of 29 bit (address label plus timestamp), this results in a required external interconnect bandwidth of 7.25 $\frac{\text{Gbit}}{\text{s}}$, assuming an optimal transmission encoding. The EXTOLL network, used for the communication implementation in this thesis offers a total bandwidth of 12 $\frac{\text{Gbit}}{\text{s}}$ (cf. Equation (8.6) in Section 8.2.2) at the FPGA's link to the network.

5.3.4 Transport Reliability

In biology, synaptic transmission is observed to be quite unreliable and up to 50 % of the arriving presynaptic pulses fail to evoke a postsynaptic response (Allen et al. 1994; Hessler et al. 1993). (Allen et al. 1994) argues that the reason for this lies in the probabilistic nature of neurotransmitter release at synapses. However, the overall neural network seems to cope well with this unreliability. It can therefore be reasoned that it is not so important to transmit every spike event reliably to a target neuromorphic synapse circuit, given that the reliability of that circuit is significantly higher than its

biological counterpart. However, this holds only true when biological networks are modelled. In the case of artificial SNN models, every single spike can be quite important and carry significant information Datta et al. 2021. In any case, it must be ensured that the dropping of events only happens stochastically and not systematically for a particular group of source- or destination neurons. Additionally, the overall drop rate of events should not become too large, such that the overall function of the modelled neural network would become disturbed.

While reliability towards the completeness of event transmission can be considered less important, reliability with respect to correctness should however be guaranteed in any case. Incorrect transmission of events could easily lead to wrong timing, resulting in large latency jitter and increased drop rates. Furthermore, incorrectly transmitted events could end up at wrong target neuron populations, disturbing the functional connectivity pattern of the modelled neural network. Therefore, reliable and correct transmission of spike events is essential to the function of a neuromorphic computing system.

According to (Karasenko 2020), the event transport between the BSS-2 ASIC and FPGA is not secured for data integrity, as the bit error rates are found to be reasonably low. For external event communication, the EXTOLL network ensures reliable transport of all data packets by strong CRC protection (cf. Section 4.1.1).

5.4 Methods for obtaining Spike Communication Quality of Service

In Section 5.3 the main aspects of Quality of Service in spike event communication were summarised and explained. This Section now presents general methods of how to reach these requirements, while in Section 5.5 some examples of existing implementations employing these methods in different ways are shortly presented.

As stated in Section 5.3.1, the transmission latency of individual spike events might have strongly model-dependent optimal values. Therefore, the final axonal delay of individual spike events should be decoupled from the actual transmission of the AER data. As long as the transmission delay through the interconnect is low enough, any axonal delay can in principle be implemented by delaying the final emission of the transported spike events to the target synapse. However, the delays are exposed to an upper limit by the physical buffer capacities at the receiving network node, as the buffer requirements will rise linearly with the implemented delay as well as the incoming aggregated event rate.

The latency constraint always has to be fulfilled together with the more important jitter constraint. As explained in Section 5.3.2, any jitter on these delays will have a direct impact on the precision of STDP learning results and general input integration of the neurons with respect to the neurons' time constants. Generally, both constraints are solved by modifying the timestamp of events at the source by adding the required axonal delay and ensuring that this delay is larger than the maximum transmission latency across the interconnection network between the source- and target neuromorphic device. Thereby, any jitter in the transmission latency can be settled out by the final buffer stage at the destination node:

$$t_a = t_e + d \tag{5.1}$$

with t_a being the resulting arrival timestamp, computed by the emission timestamp t_e at the source

and the axonal delay d. In case, one presynaptic neuron is connected to several postsynaptic neurons with different delays, this would have to be applied multiple times accordingly.
The transport bandwidth in the interconnection network is strictly limited by the technical implementation and characteristics of the interconnect. Therefore, the available physical bandwidth has to be efficiently used to guarantee as much spike event traffic with as minimal jitter as possible. In other words, spike event traffic must always have priority over timing uncritical management and configuration traffic.
As strict completeness of received events is not required by the unreliable nature of biological synaptic event reception, events that would arrive late at the target node, can and should be dropped. However, as described in Section 5.3.4, it must be ensured that those events that arrive in time are content-wise correct. This can in general be ensured by securing the network traffic at the OSI Link Layer (L.2) by means of error detection and correction mechanisms like CRC codes.

5.5 Existing Spike Communication Architectures

For transporting spike events through a digital network, there are different possible architectural design and routing strategies, regarding the addressing scheme as well as the implementation of the Quality of Service measures explained in Section 5.3 and Section 5.4. An overview on the high performance network design space has been given in Section 2.3. Some examples of network implementations for neuromorphic spike event communication from the historical context of the BrainScaleS development and the competing neuromorphic hardware system in the HBP, SpiNNaker, shall be summarised in the following subsections.
For example, (Grübl 2007; Philipp 2008) (cf. Section 5.5.1) route events according to their destination address, on a network providing time devision multiplexed circuit switching (cf. Section 2.3.3.3). In contrast, (Plana et al. 2020) (cf. Section 5.5.2) selects outgoing links based on the source address. A different approach is however proposed by (Thanasoulis 2019) (cf. Section 5.5.3), who introduces the concept of *link tags*. These are re-assigned to a spike message at each hop through the network. Thereby the routing becomes independent from the system-size.

5.5.1 Spikey and the Nathan Network

One of the first predecessors of the current HICANN-X BSS-2 ASIC was the Spikey Chip. It implemented a total of 384 Leaky Integrate-and-Fire (LIF) neurons and 98,304 synapses. It was driven by a 400 MHz system clock and operated at speed-up factors between 10^4 and 10^5. The Spikey chip was mainly developed and described in (Grübl 2007). The Spikey ASIC was mounted on a so-called Nathan module PCB together with an FPGA and local memory. 16 of these modules had been interconnected through a 2D torus network implemented on a backplane PCB (Fieres et al. 2004).
A multi-class gigabit network architecture has been developed for this platform in (Philipp 2008). That network architecture distinguished traffic in two different priority classes. A first traffic class was handled as *priority traffic* and transported through asynchronous channels with guaranteed bandwidth reservation. On the other hand, low priority traffic like for example configuration data and memory accesses were transported in a packet-based *best-effort* manner. This was achieved through

a Time Devision Multiplexing (TDM) approach, where the available physical link bandwidth was divided into time slots of a fixed length. *Priority traffic* was transported on connection-wise pre-allocated routes with reserved time slots. *Best-effort* traffic however, was transported within the unused timeslots, including unreserved ones as well as those reserved slots that were currently not used by *priority traffic*. The *priority traffic* connections were provided as *isochronous* in the terms of guaranteed throughput and minimised latency jitter.
On top of this multi-class network architecture, (Grübl 2007) developed a Neural Event Processor, handling the transmission and reception of spike events in the communication FPGA. Source events from the connected neuromorphic ASIC or the playback memory (for the concept of playback compare Section 3.2.2) are received by the event processor and converted into one or more destination events that are addressed towards a synapse row on the target chip. The creation timestamp of each source event is modified by adding the fixed delay of the respective axonal connection to obtain a release timestamp for the destination events. In order to simplify the sorting of events from different sources at a single destination, they were already delayed at the source before sending them shortly before the latest possible moment according to the expected transmission latency. This expected transmission delay was calculated dynamically based on the current network state, such that dropping of events was possible already at the sending side of a connection. This early dropping approach prevents wasting precious bandwidth for events that would arrive too late at their destination and thereby also delaying events that otherwise would have arrived on time.
In this communication approach, events are routed towards their destination, using the destination synapse driver address.

5.5.2 The SpiNNaker Network

SpiNNaker (S. B. Furber, Lester, et al. 2013; Mayr et al. 2019) is a million-core neuromorphic computing system, designed to numerically solve the dynamics of up to a billion neurons in real time. It is designed as a Globally-Asynchronous, Locally-Synchronous (GALS) system, meaning that the cores, operating on individually synchronous clocks, are interconnected through an asynchronous, clock-less network. Each SpiNNaker chip contains 18 cores, asynchronously connected to a central router unit in a star-like topology (Wu et al. 2009). Through this router unit, 48 chips are, again asynchronously, interconnected on a SpiNNaker PCB using a 2D hexagonal mesh topology. On the next level, multiple boards can be interconnected through FPGA-based high-speed serial links, called SpiNNLink (Plana et al. 2020). These links tunnel the mesh connections between the SpiNNaker boards.
Because SpiNNaker operates without a speed-up factor compared to biological realtime, (S. B. Furber, Lester, et al. 2013) argues that the electrical transmission- and switching delays, as well as any latency jitter can easily be tolerated, as they are tiny compared to the biological synaptic time constants. Therefore, events are transmitted with a pure AER code without the addition of timestamps. Large axonal delays are modelled at the destination side, while ignoring the previous physical transmission delay through the network.
As described in (Wu et al. 2009), events are transported in small packets of 40 bit to 72 bit size. Each packet contains an 8 bit management header followed by a single 32 bit event and optionally an additional 32 bit payload. Table-based routing algorithms for the SpiNNaker network are solely

based on the source neuron address. The router units can therefore support multicast communication using a Content Addressable Memory (CAM) routing table. Reliability of event transmission is ensured by error-detection and retransmission of packets. If retransmission also fails (e.g. due to congested links), packets are routed on an alternative path and ultimately dropped or stored for re-injection by a house-keeping processor core at the current position in the network.

5.5.3 BSS-1 Wafer Scale Communication

For the BrainScaleS-1 wafer-scale neuromorphic computing system, (Thanasoulis 2019) developed a dedicated event-based network. This network directly interconnects the FPGA boards that also manage the host-side communication with the neuromorphic ASIC substrate. Spike events are routed across several hops through this FPGA network. In contrast to the approach summarised in Section 5.5.1, there are no pre-reserved connections. Instead, events are routed flexibly using a packet-based approach. A new addressing scheme is proposed in the work of (Thanasoulis 2019) with respect to the routing algorithm, which is called *axonal link tags*. In this approach, events are not routed according to their complete destination synapse address, as this would scale with the total system size and thereby present a non-fixed header overhead, especially for small packets. Instead, each hop assigns link tags in its routing table, independent of the link tags at all other hops. These link tags can be kept much smaller than the whole destination address and most importantly, independent of the system size. The latter aspect is due to the fact that a neural network is mostly not all-to-all connected, meaning that populations of neurons usually do not connect to all other populations, but only a small subset.
The events' timestamps are again modified by adding axonal delays. However, in contrast to Section 5.5.1, delays can be configured for each hop individually. As hereby, the total delay of an event is no longer predictable at the source, buffering until the timestamp becomes imminent, has to happen at the destination.

5.6 Event Communication in a Packet-Based Network

Each of the system examples, summarised in Section 5.5 has proposed and realised a custom network implementation, optimised for the unique constraints and requirements of neural event communication stated in Section 5.3. However, in the scope of this thesis another approach is taken, making use of an existing, *general-purpose* network infrastructure in order to interconnect BSS-2 systems. The EXTOLL network (cf. Chapter 4) is found to be suitable for this task, as it offers a combination of high transmission bandwidth, high sustainable message rates and very low transmission latencies (cf. Chapter 4 on page 48).
In a packet-based network all information is transported in packets with a given size and destination address. The network infrastructure is responsible for the routing of packets from their source to their destination. Flow control ensures that transmitted data does not overflow the buffering capacities in the nodes on the path through the network. To route a packet to its target node, some management information has to be transported together with the payload. This header information contains at least the destination node address, as well as the size of the packet. The address is used to determine the next step on the route towards the destination from a routing table, while the size

is used for flow control purposes. While the *special-purpose* networks mentioned previously, keep packets quite minimal (typically a single event which acts both as header and payload at the same time), *general-purpose* high performance networks usually transport a lot more management information, like e.g. process identifiers, security information and intra-node addressing for DMAs. To compensate for this large amount of header-information and keep the overhead low, the amount of payload transported within a packet is usually also much larger.

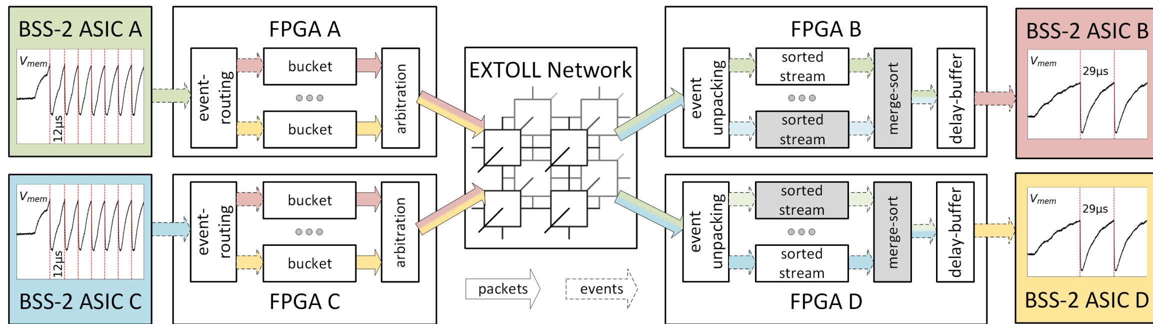


Figure 5.2: Schematic depiction of the packet-based BSS-2 event communication. The colour code exemplarily shows the target- and source nodes of packets and events. On the left, **ASIC A** and **ASIC C** produce events for both **ASIC B** and **ASIC D** where they excite target neurons to emit spikes. Consequently, both ASICs on the right side receive events from both ASICs on the left. The greyed out merge-sort blocks are not yet implemented (cf. Section 9.2 on page 194). Times are given in hardware units with a speedup factor of 10^3. This Figure is slightly modified from (Thommes, Bordukat, et al. 2022).

The main goal of this thesis is, to develop an event communication mechanism that can cope with the afore mentioned features of general purpose high performance interconnection networks. Figure 5.2 shows a schematic representation of this event communication architecture. It exemplarily shows four BSS-2 ASICs emitting spike events that are transported through the packet-based EXTOLL interconnection network. The neuromorphic nodes consist of one BSS-2 ASIC and an FPGA node (cf. Chapter 3) that is directly connected to the EXTOLL network (cf. Chapter 4). For a large network installation the favoured topology for the EXTOLL network is a 3D Torus where the FPGA nodes are connected to concentrator nodes, which are attached to the torus nodes on the respective 7th link of each EXTOLL Tourmalet ASIC node. A schematic visualisation of the envisioned large-scale network topology is shown in Figure 5.3.

As the EXTOLL routing is based upon the 16 bit destination node address in the packet header and the choice of the routing key is not customisable, spike events are aggregated into packets heading for the same target node (cf. Section 5.6.1). On the receiving side, the incoming events are first unpacked and the resulting event streams from different source nodes will be merge-sorted to form a single event stream. The individual events are then delayed until their timestamp matches the synchronised systime and transmitted down to the connected BSS-2 ASIC (cf. Section 5.6.2). As the merge-sort stage is not yet included in the implemented design, up to now only events from one source node can be received (cf. Section 9.2 on page 194). A connection scheme, where neurons from one source node connect to neurons on several destination nodes, is also possible through multicast messages in the EXTOLL network. However, these would exhibit a unique axonal delay for all receiving nodes of the multicast operation. In order to compensate for the network jitter, this

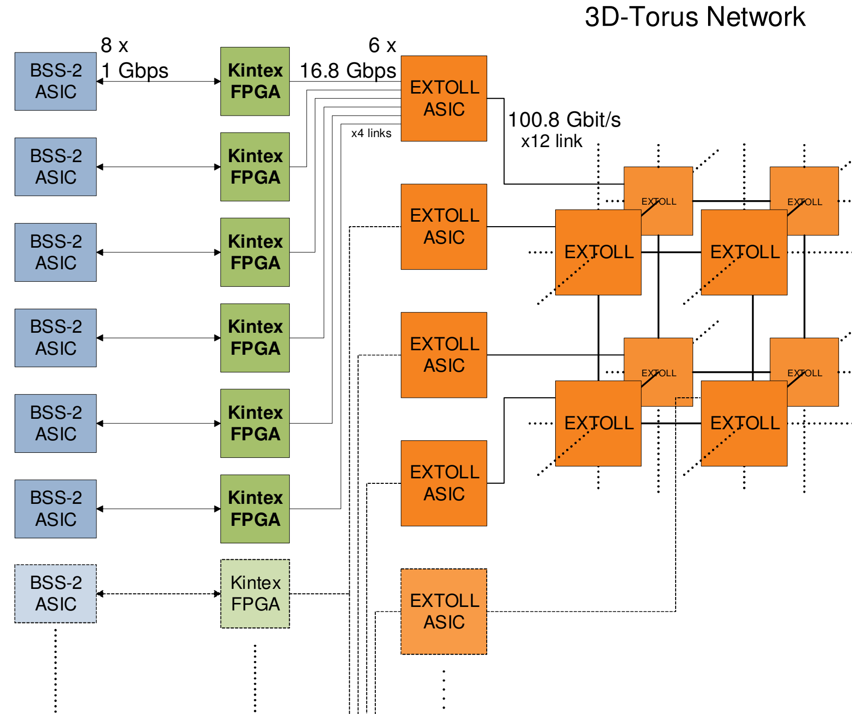


Figure 5.3: Schematic depiction of the envisioned large-scale BrainScaleS EXTOLL network. The topology will be realised in a 3D-Torus with concentrator nodes, connecting to 6 FPGA nodes. Figure modified from (Thommes 2018)

delay has to be set to the maximal distance from the source node to the destination nodes.

5.6.1 Event Aggregation

Spike events from SNNs usually carry only a small amount of information, as described in Section 5.2. Generally the size of a spike event can be expected to be much smaller than the header of the network packet transporting the event to its destination node. This poses an issue to the transmittable payload bandwidth, as a large portion of the overall network link bandwidth will be used by header information instead of actual payload data. Fortunately spike events from neuromorphic chips implementing a large SNN will often share the same target node address. Aggregating those events with a common destination would significantly increase the bandwidth efficiency. However, statistically the events sharing the same destination will not be subsequent in the event stream from a source chip. Therefore, simply transmitting consecutive events together in an aggregated packet is not a general solution to this problem. Instead, one has to sort the event stream by destinations.

Employing a traditional sorting algorithm for this task is not feasible, as it would require the pre-buffering of a large chunk of n events that would then have to be sorted with a best case algorithmic complexity of $\mathcal{O}\left(n^2\right)$ read- and write accesses. Traditional sorting is also not applicable to the problem, as the destinations do not exhibit an ordering; it is not important in which sequence packets are sent to the different destinations, apart of the events' required arrival time. Instead, the sorting operation can and has to modify the event stream *on-the-fly*, i.e. in $\mathcal{O}(n)$ complexity. This is possible by providing a number of send-buffers, each accumulating events for a specific destination. In

the following sections and chapters of this thesis these buffers will be referred to as *accumulation buckets*. This naming refers to the similarity of this approach to the *Bucket Sort* algorithm (cf. Section 5.6.1.1). The event accumulation concept is also similar to the method of *packet aggregation* that is e.g. commonly applied for efficient transport of data across the internet (cf. Section 5.6.1.2).

The assignment of buckets to a particular destination can either be statically predetermined by configuration or dynamically derived during operation. If there are more possible destinations in the output event stream, than buckets provided by the implementation, the bucket buffers have to be reused for aggregating more than one destination. This implies that a static assignment is not longer viable and the buckets will experience context switches, occurring, when the event stream presents a destination address that is not currently accumulating and there is no idle accumulation bucket available to start accumulating it. The dynamic assignment of buckets can be compared to the process of register renaming in computer architecture where hardware registers are dynamically assigned to computing instructions (cf. Section 5.6.1.3).

The time available for accumulating events into a packet depends on the modelled axonal delays of events accumulated into the respective packet. On the other hand, there should be enough time to accumulate a reasonable amount of events in order to significantly reduce the header overhead O_h which is defined as the ratio of header size s_h to the total packet size s_pkt, including the payload size s_pl:

$$O_\mathrm{h} = \frac{s_\mathrm{h}}{s_\mathrm{pkt}} = \frac{s_\mathrm{h}}{s_\mathrm{h} + s_\mathrm{pl}} \tag{5.2}$$

Analogously, the payload efficiency E_pl is defined as

$$\begin{aligned} E_\mathrm{pl} &= \frac{s_\mathrm{pl}}{s_\mathrm{pkt}} = \frac{s_\mathrm{pl}}{s_\mathrm{h} + s_\mathrm{pl}} \\ &= 1 - O_\mathrm{h} \end{aligned} \tag{5.3}$$

Consequently, the axonal delay is constrained by the model requirements as well as the required time to accumulate events and transmit the packet across the network. The axonal delay d should in any case be large enough to counteract the latency jitter, introduced by the accumulation, transmission and sorting (cf. Equation (5.1)). The delay must therefore be larger than the maximum value of the total transmission latency $\max(l_\mathrm{tot})$:

$$\begin{aligned} d &\geq \max(l_\mathrm{tot}) \\ &= \max(l_\mathrm{link} + l_\mathrm{acc} + l_\mathrm{trans} + l_\mathrm{sort}) \ . \end{aligned} \tag{5.4}$$

Thereby l_link is the transmission latency between the creation of the timestamp at the L2 interface on the HICANN-X and the accumulation buffer on the FPGA. This has to be added only once, as in the opposite direction, from the FPGA to the HICANN-X chip it is already taken care of by the receiving L2 interface on the chip. Furthermore, l_acc is the time spent to accumulate events towards a common destination, l_trans is the pure transmission latency across the network between source- and destination node, and l_sort is the time required to merge-sort different event streams at the destination (cf. Section 5.6.2).

By expressing the payload size through the size of a single event s_ev and the rate $r_\mathrm{ev}^{d^\star}$ at which events

are heading towards a respective destination $d^\star$

$$s_\text{h} = s_\text{ev} \cdot r_\text{ev}^{d^\star} \cdot l_\text{acc} \tag{5.5}$$

the accumulation time required to reduce the header overhead below a threshold o_h can thereby be derived as

$$\begin{aligned} & O_\text{h} \leq o_\text{h} \\ \Leftrightarrow \qquad & l_\text{acc} \geq \frac{s_\text{h}}{s_\text{ev} \cdot r_\text{ev}^{d^\star}} \cdot \frac{1 - o_\text{h}}{o_\text{h}} \quad . \end{aligned} \tag{5.6}$$

Analogously, the same derivation can be done for the payload efficiency E_pl being larger than a threshold e_pl:

$$\begin{aligned} & E_\text{pl} \geq e_\text{pl} \\ \Leftrightarrow \qquad & l_\text{acc} \geq \frac{s_\text{h}}{s_\text{ev} \cdot r_\text{ev}^{d^\star}} \cdot \frac{e_\text{pl}}{1 - e_\text{pl}} \quad . \end{aligned} \tag{5.7}$$

A more in-detail formal analysis of the event aggregation process can be found in Chapter 6.

5.6.1.1 Bucket Sort

The basic Bucket Sort algorithm is described in (Cormen et al. 2009). It takes an Array of n real numbers and sorts them in expected linear time. To achieve this, it first scatters them into $k = c \cdot n$ intervals, called *buckets*, where c is a constant factor. The n numbers are thereby assigned to the respective interval bucket by value. These buckets are then individually sorted, e.g. by insertion sort, before they are concatenated to form a list of now sorted numbers. The *Bucket Sort* algorithm completes in expected time $\mathcal{O}(n)$, if the intervals are chosen, such that the n numbers distribute statistically equally across them. In that case the individual sorting of the buckets will on average be a constant operation with $\mathcal{O}(1)$ complexity, as every bucket will on average only contain c numbers. There exist several variants of the *Bucket Sort* algorithm optimising different aspects of the original algorithm (Corwin et al. 2004). For example, *Counting Sort* (Cormen et al. 2009) first counts the number of elements of each key and then sorts them by direct exchange to the correct position in the output. In contrast to the original *Bucket Sort*, this requires less memory or a less complex data structure holding the data. Another variant example is *Postman's Sort* (Black 2011) where elements are scattered to the buckets by the use of hierarchical features of the element keys. This is for example used in post offices where letters are first sorted by region, then by city and lastly by street and house number.

5.6.1.2 Packet Aggregation

Packet aggregation generally describes the concept of combining individual packets to larger ones containing the data and (part of) the header information of each sub-packet. This is beneficial, if the smaller sub-packets share a common path on their routes or even the same destination. Thereby, the network routers on the shared path only need to process one packet instead of multiple ones which increases the total amount of (sub-) packets that can be transported through the network. Another

advantage is that the headers of the sub-packets can probably be combined, leading to less overall header overhead.

There are several patents claiming inventions in this area. For example (Ketcham 2004) presents a general "method and apparatus for packet aggregation in packet-based network[s]", while (Rajkumar et al. 2008) presents "packet aggregation for real time services on packet data networks". The latter patent especially applies to the case of timing-critical information like e.g. Voice over Internet Protocol (VoIP) or video conference applications. In an aggregated packet with time-delay intolerant data that part with the most strict timing constraints will dictate the timing constraint on the whole aggregated packet. Especially in wireless networks, the concept of packet aggregation is widely used to improve the performance of small packet transfers across multiple hops (K. Kim et al. 2006).

However, in contrast to these packet aggregation approaches, the event aggregation, as described in this thesis cannot make use of the full potential of merging packets only on common parts of their otherwise different routes. This is because the EXTOLL network does internally not support the aggregation and de-aggregation of packets. Therefore, events can only be aggregated to packets at the source node if they are heading for the exact same destination node.

5.6.1.3 Register Renaming

Register renaming is a common technique in processor design, used to improve the performance regarding instructions per cycle in the presence of false instruction dependencies like Write After Read (WAR) or Write After Write (WAW) respectively. For example if an instructions `i1` computes something, based on the value of a register `r1` and a second instruction `i2` computes something different and wants to store its result in `r1`, `i2` would have to wait until `i1` has finished reading from `r1` to not mangle `r1`'s result. However, this dependency is called *false*, as the result of `r2` does not at all depend on the result of `r1`. This kind of dependency can be resolved by writing the result of `i2` to another register (e.g. `r10`), i.e. renaming `r1` in the scope of `i2`. The actually used result registers are thereby dynamically assigned to the issued instructions. Following instructions, truly depending on the result of a previous one that has been subject to register renaming thereby also have to be informed about the new name of that register. (Sima 2000) gives a comprehensive review on "the design space of register renaming techniques". The main design space dimensions are specified to be the following.

- The *scope* on which the renaming of registers is applied, i.e. which type of registers are subject to renaming and dynamic assignment.
- The *layout* of the renaming buffers with respect to their type, their number and the number of read- and write ports to access them.
- The method of mapping registers to entries in the renaming buffers.
- The rate at which renamed registers can be assigned to issued instructions.

For each of these dimensions there exist several approaches, which are summarised and referenced to existing implementations by (Sima 2000).

5.6.2 Event Reception

When the spike event packets arrive at their destination, they have to be unpacked and forwarded to the neuromorphic chip. In order to ensure the correct timing of events with respect to the axonal delays and counteract any jitter, introduced by transmission or accumulation, every event has to be delayed, until its timestamp is recent with respect to the systime counter. However, as the timestamp was created and modified with respect to the source node's systime and is interpreted with respect to the target node's systime, both time counters have to be synchronised to represent a global time measure. Otherwise, the interpretation of the timestamp for event playback at the target neuromorphic node will depend on the systime offset between source- and target node. This offset will always depend on the individual startup latency and the order of system startup, and is thereby more or less random. Additionally, an asynchronous start of stimulus playback programs on different nodes of the system would introduce similar offsets in the overall spiking operation. It is therefore crucial to have a mechanism for synchronisation of a global systime counter as well as a globally synchronous experiment start. A method for this synchronisation of systimes is presented in Section 7.2 and evaluated in Section 8.6.3.

If the modelled axonal delays would be set for each neural connection individually, between source neuron and target synapse, events from incoming packets would inevitably have to be sorted at the target node before they are delayed and forwarded to the neuromorphic chip. Otherwise, events scheduled for a later point in time would block the forwarding of events that were generated later, but should arrive earlier due to a shorter requested delay. Possible implementations for this sorting problem will be concisely reviewed in Section 9.2 on page 194. However, this sorting operation can be a quite complex task with respect to execution time, as the magnitude of disorder on the event stream can be quite large and principally undefined or unknown. The said magnitude of disorder M_d is defined as the maximum backwards distance between two timestamps t_i and t_j arriving in the event stream $\mathbb{S}$

$$M_\mathrm{d} = \max_{i<j\in\mathbb{S}} [t_i - t_j] \quad . \tag{5.8}$$

The sorting latency (l_sort in Equation (5.4)) will at least be as large as the magnitude of disorder. This is because an incoming event has to be buffered until it is certain that no event with earlier scheduled arrival time can arrive anymore. Alternatively one could just *define* a sorting delay period and drop any event that arrives after this period and cannot be sorted into the stream anymore.

$$l_\mathrm{sort} \geq M_\mathrm{d} \tag{5.9}$$

However, if the delay is the same for all events between a source- and a target node, and the events are transported in order from the neuromorphic chip towards the communication interface and across the network, the event stream will also arrive in order at the target node. In this case, the sorting operation would not be strictly necessary.

However, there will generally still be several distinct event streams arriving in packets from different source nodes. These have to be merged in the correct order to form a single event stream down to the neuromorphic chip. The latency required for this merging operation again depends on the length of the incoming event packets and the magnitude of disorder M_d between the event timestamps from all the packets. The merging can notable be done with the same algorithm than the sorting operation,

as described in Section 9.2.
When having the incoming event stream sorted, the events have to be forwarded at just the right time with respect to their axonal delay and timestamp. For this, the events are stored in a FIFO buffer, from where they are extracted and forwarded down to the neuromorphic chip when their timestamp is greater or equal to the global systime. The required size of this delay buffer can be estimated by how far the events may lie in the future after the sorting operation at the destination node. By assuming, to just have received an event that lies maximally in the future, the buffer should still be able to receive a new input event at every following clock cycle until the first event can be released. Thereby, the required buffer space S_{dbuf} evaluates to the minimum total transmission latency across the network $\min(l_{\mathrm{tot}})$ (in units of clock cycles) subtracted from the configured axonal delay d at the sending side (cf. Equation (5.4))

$$\begin{aligned} S_{\mathrm{dbuf}} &\geq d - \min(l_{\mathrm{tot}}) \\ &= d - \min(l_{\mathrm{link}} + l_{\mathrm{acc}} + l_{\mathrm{trans}} + l_{\mathrm{sort}}) \ . \end{aligned} \tag{5.10}$$

6 Formal Analysis of Event Aggregation

As described in Section 5.6, it is necessary to accumulate events to larger packets per destination to mitigate the transmission overhead induced by the packet header in a general purpose network. This accumulation can be done by so-called bucket buffers. These are individually assigned to a particular network destination and collect all events from the incoming event stream heading to this particular destination.

In this Chapter, the focus lies on the question of how performant this accumulation process can be, depending on statistical properties of the incoming event stream. The key performance indicators are thereby defined by the following questions:

- How many events can on average be accumulated in a packet, until it can (or has to) be sent across the network?
- How long does it on average take to accumulate events to a packet until it has to be sent across the network?

A generic example for a partitioned neural network, distributed across several system units is depicted in Figure 6.1. A system unit thereby consists of a neuromorphic chip and a communication FPGA (cf. Figure 5.2). Each neuromorphic chip implements a part of the overall neural network model, each containing several populations of neurons. Populations on source units are thereby connected to populations on other destination units. It should be noted that the mapping of where a population is placed in the system is a degree of freedom and can be leveraged for optimisation. Based on the average firing frequency (cf. Section 2.2.3) of the individual populations, some destination units are more likely being addressed than others, which is indicated in the Figure by exemplary numbers.

The same model example is depicted in Figure 6.2, but now from the event communication perspective, showing the bucket buffers. The probability numbers now represent the time share of the particular destinations on the accumulation process of each bucket. Every time, a bucket's input event stream presents a destination not matching the current accumulation, a context switch is necessary, defining a *destination conflict* (cf. C.3 in Section 6.1.1). It should again be noted that the mapping of which bucket is to accumulate a specific destination is another degree of freedom in the process, open for different assignment strategies which are explained in Section 6.1.2. This can be leveraged for optimisation, which is explained in Sections 6.3.5 to 6.3.6. The contents of this Chapter have been presented on a poster at the 7th HBP Student Conference in January 2023 (cf. Thommes, Grübl, et al. 2023).

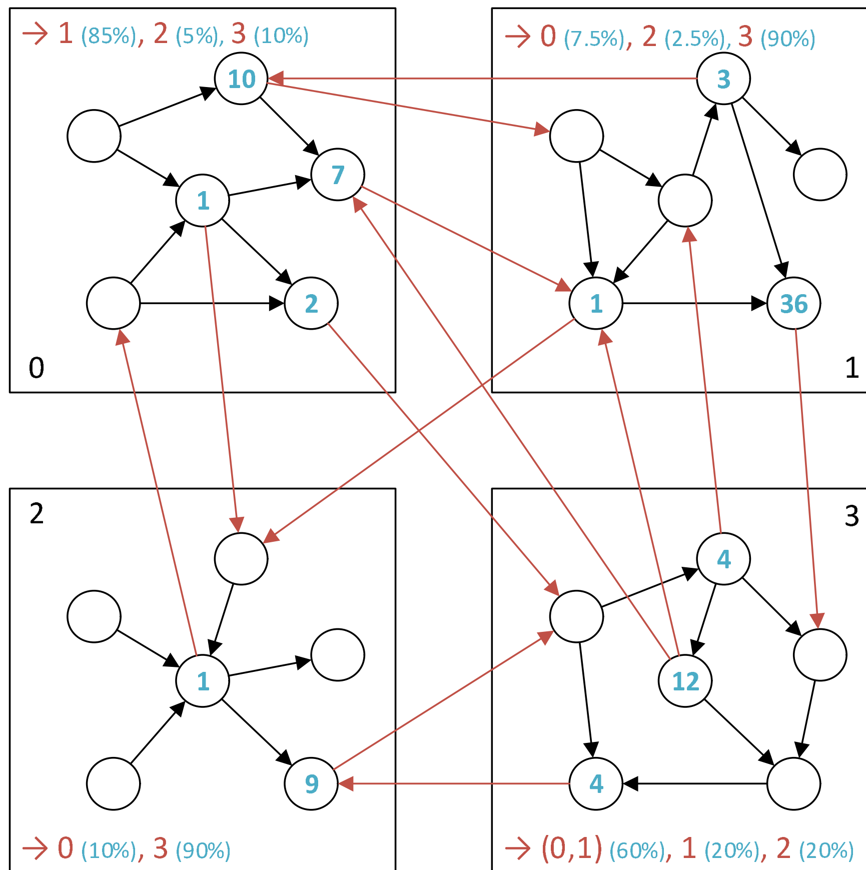


Figure 6.1: Generic example of a partitioned neural network model, distributed over four neuromorphic system units 0 to 3, from the chip-perspective. Circles represent (populations of) neurons, the blue numbers within give exemplary firing frequencies in arbitrary units. **Red arrows** depict axonal inter-chip connections. (Percentage numbers) are given for the share of outgoing events to the respective **target-chip (multicast)** from the respective chip's perspective. Multicast connections are indicated by **several arrows** leaving the same source neuron (population) and **(combined, target, ids)**.

6.1 Accumulation Buckets

The characteristics of the accumulation process shall now be formally defined, in order to specify the design space of a bucket buffer.

6.1.1 Packet Completion Criteria

Any buffering in the data path to the network will inevitably introduce more latency to the transmission. Therefore it is desirable to only accumulate events as long as necessary until sending them across the network. This creates a trade-off condition between the utilisation of the available packet-space which is directly connected to the header overhead, and the induced latency by buffering and accumulating the events. In order to avoid large latencies, the bucket should then immediately be marked as free and request permission to transmit its accumulated contents. During the time until sending permission is granted, the bucket should be able to already accumulate new contents for the next packet. There are three conditions under which an accumulated packet is eligible for sending

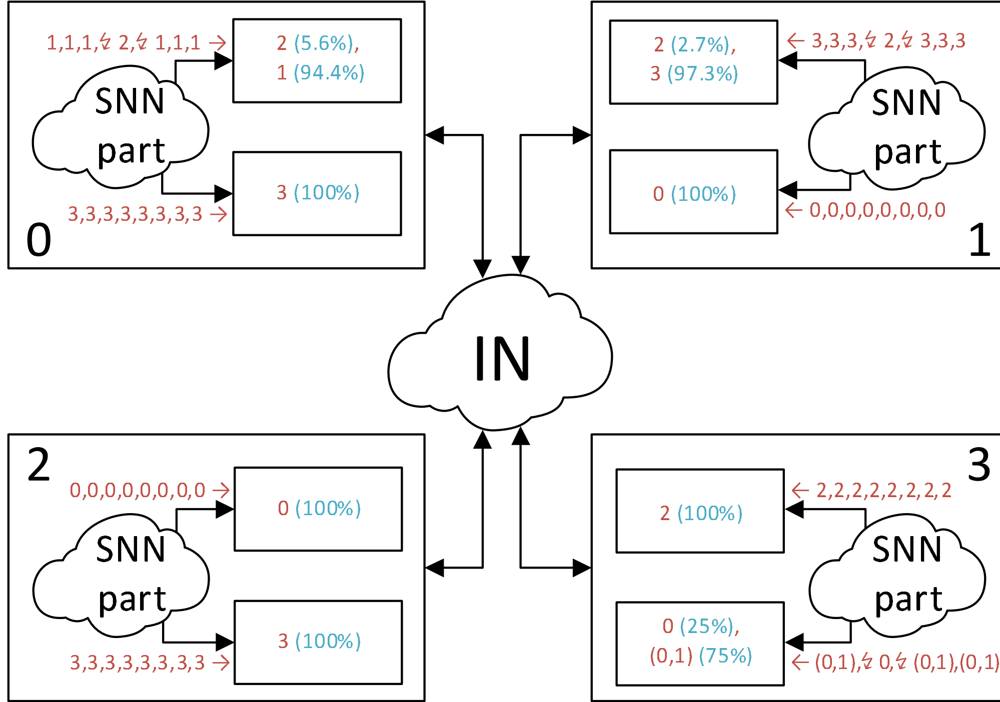


Figure 6.2: Generic example of a partitioned neural network model, distributed over four neuromorphic system units 0 to 3, from the FPGA-perspective. Events emitted by the SNN parts are accumulated in buckets. The **percentage numbers** give the share of the particular bucket, accumulating events for the respective **destination**. Exemplary **sequences of destinations** are given at the bucket inputs, marking context switches (destination conflicts) with ϟ **symbols**.

to the network, which are listed below:

C.1 Packet Full: Network packets are always limited in their size to a certain Maximum Transmission Unit (MTU) given by the implementation of the network infrastructure. When the amount of accumulated events k reaches the maximum number fitting in the packet K, it is considered full and has to be sent.

$$k = K \tag{6.1}$$

C.2 Event Timeout: As described in Section 5.2, each spike event e carries a timestamp t_c stating the time of creation at the source neuron. This creation timestamp is converted to an arrival timestamp t_a, by adding the modelled axonal delay t_d (cf. Equation (5.4)). The resulting timestamp now states the required time of arrival at the destination neuron.

$$t_a = t_c + t_d \tag{6.2}$$

This conversion allows to minimise the transmission latency-jitter, as required in Section 5.3. With a worst-case estimation of the network latency between the source- and destination-node (including the time required to unpack and merge-sort events at the destination), there is now a constraint to the latest point t in time to transmit an event to the network for it to arrive just

in time at its destination neuron. When the most critical event, accumulated in the current bucket b reaches this threshold time t_{th} (including a safety margin for the network arbitration), the packet has to be sent.

$$t \geq \min_{e \in b} t_a(e) - t_{th} \tag{6.3}$$

This condition can be simplified by defining a timeout period l_{aho} that the bucket can use for accumulating events and calculating the delay t_d accordingly (cf. Equation (5.4)).

C.3 Destination Conflict: When there are more possible destinations than buckets, the individual buckets have to be reused for more than one destination. Every time there is an event e with destination d_e at the input, requesting assignment to a bucket b that is already accumulating events for another destination d_b, the bucket has to be flushed and relabelled in order to now accumulate events for the new destination d_e.

$$d_e \neq d_b \tag{6.4}$$

This is also depicted in Figure 6.2.

6.1.2 Strategies for Destination Assignment

The assignment of destinations to the buckets can be permanent in the case when there are enough buckets for all occurring destinations. Otherwise, when there are more destinations than buckets, the assignments become ambiguous, i.e. more than one destination will be assigned to the individual buckets. This ambiguous assignment can be created dynamically or statically.

A **static strategy** has a fixed assignment pattern available for any events at the input. Buckets are then requested and interrupted as necessary, according to that existing assignment pattern. In general, the static assignment of an event e to a bucket b will depend only on the event itself, particularly on its source neuron address.

$$b = f(e) \tag{6.5}$$

Thereby, each event is uniquely assigned to a single bucket.

In contrast, a **dynamic strategy** at first attempts to select a free bucket if an arriving event's address is not yet assigned to a bucket. If there is no free bucket, one has to be interrupted with a destination conflict (cf. condition Equation (6.4)). In general, the dynamic assignment of an event e to a bucket b will depend on the overall state of the bucket system S_B.

$$b = f(S_B) \tag{6.6}$$

Unlike with static strategies, the selection of the conflicted bucket is not fixed, but rather dynamic such that any bucket could be eligible. Also, as this decision depends on the state of the bucket system, it will generally lead to a different result each time brought about.

6.1.2.1 Static Assignment Strategies

AS.1 Modulo Operation: Assign the event with destination d to the bucket

$$b = d \bmod B \tag{6.7}$$

where B is the total number of available buckets. Hereby every bucket b is dedicated to a fixed number D_b of destinations

$$D_b = \begin{cases} \lceil \frac{D}{B} \rceil & \text{if } b \leq D \bmod B \\ \lfloor \frac{D}{B} \rfloor & \text{else} \end{cases} \tag{6.8}$$

where D is the total number of occurring destinations.

AS.2 Lookup Table: Assign the event with destination d to the bucket b retrieved from a lookup table L

$$b = L(d) \tag{6.9}$$

Hereby the number of destinations D_b, a bucket is dedicated to, can be determined by counting the occurrences of the particular bucket id in the lookup table:

$$D_b = \sum_{d \in L} \mathbb{1}_{\{L(d)=b\}} \tag{6.10}$$

6.1.2.2 Dynamic Assignment Strategies

AS.3 Round Robin Arbitration: Keep a pointer b_{i-1} to the last reassigned bucket. Assign the new event with destination d_e to the bucket

$$b_i = \begin{cases} b_1 & \text{if } b_{i-1} = b_B \\ b_{i-1} + 1 & \text{else} \end{cases} \tag{6.11}$$

if there is no bucket already accumulating the destination $d_b = d_e$. Thereby b_B is the last bucket available, i.e. the first case implements a wrap-around condition. Hereby every bucket is relabelled in strict rotation and fairness is ensured amongst all buckets.

AS.4 Random Assignment: Assign the new event to the a randomly (with uniform distribution) chosen bucket. Hereby fairness amongst all buckets is ensured on average.

AS.5 Fullest Bucket First: Assign the new event to the bucket which is closest to condition C.1:

$$b = \left\{ b \in S_B \,\middle|\, \forall b' \in S_B \quad k_{b'} < k_b \right\} \tag{6.12}$$

Hereby, those buckets are selected that would be flushed soon anyway due to their individual filling level.

AS.6 Next Event Timeout First: Assign the new event to the bucket which is closest to condi-

tion C.2:

$$b = \left\{ b \in S_B \;\middle|\; \forall b' \in S_B \quad \min_{e \in b'}(t_a - t_{th}) > \min_{e \in b}(t_a - t_{th}) \right\} \tag{6.13}$$

Hereby, those buckets are selected that would be flushed soon anyway due to their individual event timeout.

6.1.2.3 Comparison of Assignment Strategies

Both strategies AS.5 and AS.6 aim to optimise the utilisation of buckets before they are relabelled. However, there are cases, where this optimisation will not be of much gain. This will be the case, when the viewed selection criterion does not differ much across the buckets, i.e. when all buckets are similarly full or similarly near their timeout. In particular, this might happen if the distribution of events exhibits lots of different destinations (considerably more destinations than available buckets) with relatively similar probabilities of occurrence and modelled delays, respectively. In that case those strategies will converge towards Round Robin (AS.3), as the first bucket assigned will also have had the most chances to accumulate an event or the longest delay respectively.

Static strategies can be reckoned to be generally better suited for optimising the bucket usage than dynamic strategies. This is because with static strategies only events, preassigned to a bucket can trigger the conflict condition C.3 for that particular bucket. In contrast, with dynamic strategies any event can potentially trigger a conflict condition for every particular bucket. This will of course only make a difference, when the distribution of destination addresses is not uniform, i.e. some destinations are more likely to occur than others (cf. Section 6.3.1). With a static strategy, the more frequent destinations will only disturb the accumulation process in a subset of buckets, while with a dynamic strategy they will eventually disturb every bucket. Among the static strategies AS.1 to AS.2 the Lookup Table strategy (AS.2) is the most flexible one, as it can be directly adjusted by the user to the modelled distribution of destinations. An example for such a static assignment was already presented in Figures 6.1 to 6.2 on pages 76–77.
However, there might be cases, where the probability distribution of destinations is not known to the user, as the activity of the individual neuron populations might be unknown, making a static assignment impossible to optimise. In this case, a dynamic assignment strategy might be more helpful by doing the optimisation on-the-fly.

To make a more quantitative analysis on the eligibility of the described assignment strategies and to develop practically useful constraints for design parameters as e.g. the implemented number of buckets, the following sections follow two major approaches. Section 6.2 looks at a mathematical description of the accumulation problem featuring a Markov chain model, while in Section 6.4 the accumulation process is directly simulated, regarding the different constraints.
Both approaches will assume infinite accumulation buffer space and no time limit for delivery of the buffered events, so to derive the wanted design constraints. The conditions C.1 and C.2 can therefore be neglected in the first approach. The effect of these conditions can then be implied from the analysis result.

6.2 Mathematical Analysis

The question of interest is about the number of events that are expected to be accumulated in a bucket until it is relabelled by a conflicting event. This corresponds to the degree to which the accumulation will be limited by condition C.1. Additionally the corresponding expected accumulation time is of interest, as the events also have to be delivered to their destination in time, corresponding to the degree of limitation by condition C.2. Of course the latter also depends on the modelled delay. However the delay itself is not completely free to the modeller, as it has to be large enough to compensate the network-, accumulation-, and merging jitter (cf. Section 5.3 and Section 5.6).
The following sections will discuss, how these expectation values for accumulation length and time, as well as their related distributions can be mathematically derived.

6.2.1 Accumulation as Markov Chain

The process of event accumulation in a particular bucket can be modelled as a Discrete-Time Markov chain. A Markov chain is defined as a process, where the future state only depends on the present state and explicitly not on the past states (Markov Property, cf. Privault 2018).

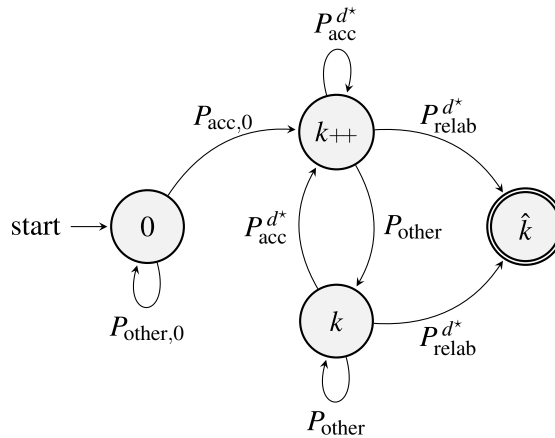


Figure 6.3: State graph of the bucket accumulation Markov chain.

The state transition graph of the accumulation process is depicted in Figure 6.3. The process starts with an accumulation level of 0 events (state (0)). From this state, the bucket will accumulate an incoming event (state (k++)) if it is dedicated to its particular destination d (in case of a static assignment strategy) or it's the buckets turn to take the event (in case of a dynamic assignment strategy). This will happen with a probability of $P_{\text{acc},0}$. Otherwise the bucket will not accumulate the event and stay in the start state (0). When the bucket has already accumulated at least one event (state (k++) or (k)), the probability $P_{\text{acc}}^{d^\star}$ to accumulate another one is slightly different, as now only those events with destination address matching the current bucket label will be accumulated. If the event belongs to another bucket with probability P_{other}, it will not be handled by this bucket and it goes to or stays in state (k). Finally, there is also a probability $P_{\text{relab}}^{d^\star}$ that the event's address is assigned to the bucket, but the current label does not match (condition C.3). In this case the bucket has to be relabelled and enters the absorbing state ($\hat{k}$). This state ends the accumulation process and the bucket content is flushed out to the network. Now the accumulation will start over again with the

new event's destination at the initial state $\textcircled{0}$.

When only evaluating the number of accumulated events and thereby the size of the network packet, represented by the number of returns to the state $\textcircled{k{+}{+}}$, the rate at which they arrive at the bucket is not relevant. A state transition through the Markov graph in Figure 6.3 will then always occur in that instant, when an event is presented at the input.
The second parameter of interest is the time until the bucket is flushed and the network packet is sent. This time can be calculated by determining the number of steps through the graph until reaching the final state $\textcircled{\hat{k}}$. Now the input event-rate cannot be neglected anymore. One possible solution to take the rate into account is to estimate the mean time between two events (cf. Section 2.2.3) and multiply that to the expected number of steps through the graph. Alternatively one can change the transition policy to having a state transition in regular intervals and include the probability for having a spike event in that interval into the transition probabilities between the particular states. Section 6.2.2.6 will elaborate the details of these two approaches.

For this Markov chain process with the state vector

$$\mathbb{S} = \left[\textcircled{0}, \textcircled{k{+}{+}}, \textcircled{k}, \textcircled{\hat{k}}\right] \tag{6.14}$$

a transition matrix can be defined by collecting the probabilities for transitioning to state $\textcircled{j}$ (column index) when starting at state $\textcircled{i}$ (row index) as follows:

$$P_{i,j} = \begin{pmatrix} P_{\text{other},0} & P_{\text{acc},0} & 0 & 0 \\ 0 & P_{\text{acc}}^{d^\star} & P_{\text{other}} & P_{\text{relab}}^{d^\star} \\ 0 & P_{\text{acc}}^{d^\star} & P_{\text{other}} & P_{\text{relab}}^{d^\star} \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.15}$$

The probability of being in a state $\textcircled{j}$ at some point in time can then be expressed by

$$\mathbb{P}(Z_n = j) = \sum_{i \in \mathbb{S}} P_{i,j} \mathbb{P}(Z_{n-1} = i), \qquad i \in \mathbb{S} \tag{6.16}$$

and depends on the probabilities to be in one of the other states before and the respective transition probabilities from those other states. Equation (6.16) represents a matrix-vector multiplication where the transition matrix is multiplied on the vector of occupation probabilities for the initial state.
This can be generalised to the probability of being in state $\textcircled{j}$ after n steps when starting in state $\textcircled{i}$ which can be computed by evaluating the n'th power of the transition matrix:

$$[\mathbb{P}(Z_n = j \mid Z_0 = i)]_{i,j \in \mathbb{S}} = \left[[P^n]_{i,j}\right]_{i,j \in \mathbb{S}} \tag{6.17}$$

Again, by multiplying the vector of initial occupation probabilities one gets the resulting occupation probability vector after n steps.

6.2.2 Derivation of Transition Probabilities

The particular transition probabilities $\left\{P_{\text{acc},0}, P_{\text{other},0}, P_{\text{acc}}^{d^\star}, P_{\text{other}}, P_{\text{relab}}^{d^\star}\right\}$ in Equation (6.15) are depending on the probability distribution and assignment strategy of the occurring destinations. In the following, the derivation for each of them will be discussed, based on these prerequisites.

6.2.2.1 Accumulating the first event ($P_{\text{acc},0}$)

Equation (6.18) calculates the probability $P_{\text{acc},0}(b)$ for accumulating the first event in the bucket b through summation over all possible destinations d.

$$\begin{aligned} P_{\text{acc},0}(b) &:= \sum_d P_{\text{acc},0}(d,b) \\ &= \sum_d P(d) \cdot P(d \to b) \end{aligned} \tag{6.18}$$

For each destination, the probability $P_{\text{acc},0}(d,b)$ consists of the product of the probabilities $P(d)$ of occurrence for the particular destination d (the probabilities given in Figure 6.1) and $P(d \to b)$ for that destination being assigned to the considered bucket b. For static assignment strategies $P(d \to b) \in \{0,1\}$ always holds true, whereas for dynamic strategies $P(d \to b) \in [0,1]$ in general and $P(d \to b) = 0$ if d is already assigned to another bucket b'.

6.2.2.2 Not accumulating an input event (P_{other})

The probability $P_{\text{other}}(b)$ that the currently presented event destination is meant for another bucket Equation (6.19) is again calculated by summation over all possible destinations d and inverting the respective assignment probability.

$$\begin{aligned} P_{\text{other}}(b) = P_{\text{other},0}(b) &:= \sum_d P_{\text{other}}(d,b) \\ &= \sum_d P(d) \cdot (1 - P(d \to b)) \end{aligned} \tag{6.19}$$

$$P_{\text{other},0}(b) = 1 - P_{\text{acc},0}(b) \tag{6.20}$$

$$P_{\text{other}}(b) = 1 - P_{\text{relab}}^{d^\star}(b) - P_{\text{acc}}^{d^\star}(b) \tag{6.21}$$

The equivalence of P_{other} and $P_{\text{other},0}$ in Equation (6.19) is only given for static assignment strategies, as explained later in Section 6.2.2.5. However, the equivalence of them both to the right part of Equation (6.19) as well as Equation (6.20) and Equation (6.21) must always hold true as the transitions leaving a particular state in Figure 6.3, i.e. rows in Equation (6.15), must sum up to 1 in order for a transition to occur with total certainty. For a mathematical derivation that this condition holds true with the definitions presented here, cf. Appendix A.3.1 and Appendix A.3.2.

6.2.2.3 Accumulating an input event ($P_{\text{acc}}^{d^\star}$)

The probability $P_{\text{acc}}^{d^\star}(b,t)$ for further accumulation of an event in an already labelled bucket in Equation (6.22) generally depends on the particular address $d^\star$ that was assigned to the considered bucket

in the first accumulation step.

$$P_{\text{acc}}^{d^\star}(b) := P(d^\star) \cdot P(d^\star \to b) \qquad (6.22)$$
$$= P(d^\star)$$

As this only accepts events to the already labelled destination, the calculation also only contains one summand from Equation (6.18). The assignment probability for accumulation of $d^\star$ is $P(d^\star \to b) = 1$, as to allow for event accumulation in the first place. Of course when one of the conditions C.1 to C.3 is met, the bucket is flushed and starts over accumulating a new first event with probability $P_{\text{acc},0}(b)$, depending on the pursued assignment strategy.

6.2.2.4 Handling a destination conflict ($P_{\text{relab}}^{d^\star}$)

Finally, the probability $P_{\text{relab}}^{d^\star}(b,t)$ for having to relabel the considered bucket can be calculated the same way as $P_{\text{acc},0}(b)$ but excluding the assigned destination $d^\star$ from the sum in Equation (6.23).

$$P_{\text{relab}}^{d^\star}(b) := \sum_{d \neq d^\star} P_{\text{acc},0}(d,b)$$
$$:= \sum_{d \neq d^\star} P(d) \cdot P(d \to b) \qquad (6.23)$$
$$= 1 - P_{\text{other}}(b) - P_{\text{acc}}^{d^\star}(b) \qquad (6.24)$$
$$= P_{\text{acc},0}(b) - P_{\text{acc}}^{d^\star}(b) \qquad (6.25)$$

Again, from the total certainty constraint in the state-transition matrix of Equation (6.15), one can formulate Equation (6.24). Equation (6.25) simply derives by either substituting $P_{\text{other}}(b)$ in Equation (6.24) or by identifying the excluded term in Equation (6.23) with $P_{\text{acc}}^{d^\star}$.

6.2.2.5 The effect of dynamic assignment

One must note that for dynamic assignment strategies, the assignment probability $P(d \to b) \equiv P(d \to b,t)$ will generally change with every transition and also depend on the other buckets in the system. Especially as $P(d^\star \to b) \equiv 1$ after the initial assignment has happened, $P_{\text{acc}}^{d^\star}(b)$ is not equal to the contribution term $P_{\text{acc},0}(d^\star,b)$ to $P_{\text{acc},0}(b)$. In order to save the validity of the matrix Equation (6.15) under summation over its rows, Equations (6.20) and (6.21) on the preceding page must both hold true. With dynamic assignment, this will generally lead to $P_{\text{other}} \neq P_{\text{other},0}$, which is again encompassed in Equation (6.19) by $P(d \to b,t)$ now depending on the state of the bucket.
Therefore Equation (6.25) is not valid for dynamic assignment. For static strategies however, this dependency on the discrete time step is constant and the latter equations can be applied. In general, when dynamic strategies are used, these transition probabilities change with every transition and therefore have to be constantly re-evaluated. Because of this, Equation (6.17) is no longer applicable and the following analysis of accumulation length and time, which is completely based on it does not hold true for dynamic assignment. Therefore in that case, the accumulation length and time for dynamic assignment have to be determined by simulation, as described in Section 6.4.

6.2.2.6 Modifications for accumulation time analysis

As already mentioned in Section 6.2.1 on page 82, the rate at which events are presented at the input to the accumulation system has to be taken into account when evaluating the accumulation time later in Section 6.2.4. When assuming an instantaneous event rate $r(t)$ that is constant over time, the event statistics at the input can be modelled as a homogeneous Poisson process as summarised in Section 2.2.3. This is especially also valid for multiple independent Poisson neurons contributing to the input event stream, as the sum of two independent Poisson processes again yields a Poisson process with the new rate being the sum of the particular rates (cf. Appendix A.2.2 for a derivation). From Equation (2.4), one can derive the mean time between two spike events (ISI) at the input as

$$\langle\tau\rangle = \int_0^\infty \tau p(\tau) d\tau = \frac{1}{r} \tag{6.26}$$

with a variance of

$$\sigma_\tau^2 = \int_0^\infty \tau^2 p(\tau) d\tau - \langle\tau\rangle^2 = \frac{1}{r^2} \,. \tag{6.27}$$

This can now simply be multiplied Inter Spike Interval with the number of steps through the Markov graph (Figure 6.3) obtained from Section 6.2.4.
Alternatively one can derive the probability P_{rate} for an event to occur within the regular time interval $(t_0, t_0 + \tau)$ at which the state transitions through the Markov graph will then occur. This leads to the rate probability P_{rate} being described as

$$P_{\text{rate}}(r, \tau) = 1 - e^{-r\tau} \tag{6.28}$$

However, with a homogeneous Poisson process, there can generally be more than one event per clock cycle τ, as the probability for a certain number of n events in the interval $(t_0, t_0 + \tau)$ is described by

$$P_{\text{rate}}^n(r, \tau) = e^{-r\tau} \frac{(r\tau)^n}{n!} \quad . \tag{6.29}$$

To solve this problem, every additional event exceeding one per clock cycle is discarded from the process. With Equation (6.29) the rate probability given in Equation (6.28) can be identified as the cumulative probability for at least one event to occur in the given interval:

$$P_{\text{rate}}(r, \tau) = 1 - P_{\text{rate}}^0(r, \tau) \tag{6.30}$$

With this event probability, the transition probabilities $\left\{P_{\text{acc},0}, P_{\text{other},0}, P_{\text{acc}}^{d^\star}, P_{\text{other}}, P_{\text{relab}}^{d^\star}\right\}$, defined above can now be modified. In detail, these modifications are the following:

$$P_{\text{acc},0}^\star := P_{\text{rate}} \cdot P_{\text{acc},0} \tag{6.31}$$

$$P_{\text{other},0}^\star := (1 - P_{\text{rate}}) + P_{\text{rate}} \cdot P_{\text{other},0} \tag{6.32}$$

$$P_{\text{acc}}^{\star\, d^\star} := P_{\text{rate}} \cdot P_{\text{acc}}^{d^\star} \tag{6.33}$$

$$P_{\text{relab}}^{\star\, d^\star} := P_{\text{rate}} \cdot P_{\text{relab}}^{d^\star} \tag{6.34}$$

$$P_{\text{other}}^\star := (1 - P_{\text{rate}}) + P_{\text{rate}} \cdot P_{\text{other}} \tag{6.35}$$

Mostly, the transition probabilities are simply proportionally scaled with the rate probability. However, P_{other} also includes the probability that there is no event at the input, corresponding to the inverse event-rate. A proof that these modifications do not invalidate the transition matrix Equation (6.15) can be found in Appendix A.3.3 and Appendix A.3.4.

If the event rate r, in contrast to our assumption above, is not constant over time, the first approach of simply multiplying the mean ISI breaks apart, as now there is no such mean ISI anymore. Instead the second approach has to be used where the rate probability $P_{\mathrm{rate}}(t)$ is now a function of time. This has again the same effect, as a dynamic assignment strategy would have which is described in Section 6.2.2.5. One can choose the time interval width to be small enough in order for the rate to be approximately constant as also done by (Heeger 2000). Then, for each time step one has to recompute the transition probabilities and re-evaluate the accumulation time.

However, as mentioned before, for the accumulation length in Section 6.2.3, these considerations are not of relevance as the Markov transitions are not evaluated at a regular clock, but exactly at those points in time, when an event is presented at the input.

6.2.2.7 Averaging over the first accumulated events

As noted before, the probability for further accumulation $P_{\mathrm{acc}}^{d^\star}(b)$ and for relabelling the bucket $P_{\mathrm{relab}}^{d^\star}(b)$ depend on the particular destination $d^\star$ that was assigned to the bucket in the first step. Therefore, for each destination that might be assigned to the bucket b, there is a different transition matrix. To account for this fact, the following analysis for the accumulation length and time has to be done separately for each of them. The resulting expectation values must then be averaged by weighted summation with the relative probabilities of occurrence for the respective destinations

$$\overline{\mathbb{E}}(b) = \sum_{d^\star} P_{\mathrm{norm}}(d^\star, b) \cdot \mathbb{E}(b) \qquad \text{with} \tag{6.36}$$

$$P_{\mathrm{norm}}(d^\star, b) = \frac{P(d^\star)}{\sum_{d, P(d \to b) \neq 0} P(d)} \quad . \tag{6.37}$$

6.2.3 Accumulation Length

The accumulation length N_{acc} of a bucket is defined as the number of events accumulated to a single network packet before sending. It will be decisively determined by one of the conditions C.1 to C.3 on page 77. Which of those will be the first one to apply and when this will happen, largely depends on the transition probabilities between the states of the Markov chain model (Figure 6.3). In particular, when taking the conditions C.1 and C.2 into account the probabilities will not be constant, but rather depend on the variables k of currently accumulated events and t as the current time. They also depend on the parameters K describing the available accumulation space as well as the minimum event timestamp t_a and transmission threshold t_{th} according to Equation (6.1) and Equation (6.3).

While C.1 just strictly limits the accumulation length as a static constraint, independent of the incoming events, C.2 may pose a dynamic constraint, meaning that the accumulation length now also may depend on the content, namely the timestamp, of incoming events. The nearest static approximation to this constraint would be a timeout counter starting when the first event is accumulated

and limiting the overall time that this first event is allowed to be kept in the bucket. The approximation is exact if the first event has the lowest arrival timestamp amongst the subsequent events. This will trivially be fulfilled for chronologically sorted event streams where every successive event has a higher timestamp than the preceding one.

As it is rather complicated to evaluate all these constraints together without a detailed simulation, the problem will here be approached step by step. At first, only condition C.3 is considered to evaluate the accumulation length (this Section 6.2.3) and time (Section 6.2.4) under the assumption of infinite accumulation space ($K = \infty$). In reality, K will be given by the Maximum Transmission Unit (MTU) (cf. Appendix B.2) of a network packet, as well as the size and packing of events to be transported in those packets (cf. Section 7.3.3). Together with a given probability distribution of the occurring event destinations, this will allow to optimise the assignment strategy of incoming events to a number of buckets. The assignment should be chosen such that the expected number of accumulated events before a conflict occurs is larger or near the available packet space.

6.2.3.1 Expected Accumulation Length

In Figure 6.3 the accumulation length can be identified as the number of visits in state $\textcircled{k{+}{+}}$ until absorption in $\textcircled{\hat{k}}$ occurs. According to Equation 5.4.7 of (Privault 2018) the expected number of Returns R_j to state j, which is defined as

$$R_j := \sum_{n=1}^{\infty} \mathbb{1}_{\{X_n = j\}} \tag{6.38}$$

can be computed with a geometric series as

$$S_{ij} := \mathbb{E}\left[R_j \mid X_0 = i\right] = \sum_{n=1}^{\infty} \left[P^n\right]_{i,j} \tag{6.39}$$

which is equivalent to

$$S_{ij} = -\mathbb{1}_{\{i=j\}} + \left[(I_d - P)^{-1}\right]_{i,j} \tag{6.40}$$

by using the geometric series

$$\sum_{k=0}^{\infty} r^k = \frac{1}{1-r}, \qquad -1 < r < 1\,. \tag{6.41}$$

Thereby the $-\mathbb{1}_{\{i=j\}}$ in Equation (6.40) encodes the fact that for the diagonal entries the first visit in state $\textcircled{i}$ is not counted as a "return". It should be noted that the matrix $(I_d - P)$ becomes singular, i.e. non-invertible if the Markov chain contains absorbing states as in Equation (6.15). The solution to this problem, as (Sigman 2016) points out, is to only consider the transient states T in Equation (6.40). The transition matrix including absorbing states will generally have the form

$$P = \begin{pmatrix} P_T & P_A \\ 0 & I_d \end{pmatrix}. \tag{6.42}$$

Here P_T denotes the sub matrix only containing transient states and P_A contains the transition probabilities for entering the absorbing states. The lower right corner of P is the identity matrix for pure absorbing states, i.e. there are no transitions between the recurrent states. This part is the reason for $(I_d - P)$ being singular. Because the sub matrix P_A does not influence the P_T part under potentiation of the complete matrix P^n, it is valid to only use $(P_T)^n$ in Equation (6.39). In the case when there is only one absorbing state, P_A resembles a column vector and I_d shrinks to a scalar 1.
With this argument and baring in mind the dependency of the transition matrix of the accumulating destination $d^\star$, Equation (6.40) translates to

$$S_{ij}(d^\star) = -\mathbb{1}_{\{i=j\}} + \left[(I_d - P_T(d^\star))^{-1} \right]_{i,j} \tag{6.43}$$

and with Equation (6.37) one gets

$$\boxed{S_{ij} = \sum_{d^\star, P(d^\star \to b) \neq 0} P_{\text{norm}}(d^\star, b) \cdot S_{ij}(d^\star)} \quad . \tag{6.44}$$

Evaluating S_{ij} for $i = 0$ and $j = 1$ now leads to the expectation value for the number of returns to state $\textcircled{k_{++}}$ while starting in state $\textcircled{0}$ which equals the expected number of accumulated events in the bucket, given the transition probabilities for the accumulation process (cf. Figure 6.3 and Equation (6.15)).

$$\boxed{\mathbb{E}(N_{\text{acc}}) = S_{0,1}} \tag{6.45}$$

6.2.3.2 Distribution of Accumulation Lengths

The expectation value computed before, might actually not quite be the information of interest, as it represents a weighted mean value. One could for example imagine a situation, where most of the packets are minimally accumulated while in some rare cases very high accumulation lengths occur. In this case the few high accumulations could middle out or even dominate the expectation value, while the common accumulation is much worse.

An example for a game with inappropriate expectation value is the so-called St. Petersburg Paradox (Peterson 2022). It defines a game where a fair coin is flipped until it shows heads for the first time. The reward is doubled with each flip, so the player will win 2^n€ with n being the number of flips executed. The expectation value for this game is

$$\sum_{n=1}^{\infty} \left(\frac{1}{2}\right)^n \times 2^n = \infty.$$

So theoretically the player should pay any stake to start the game, as the expected reward is infinitely high. But actually in most cases the player will only win a very low amount of money, as the high rewards are so unlikely. Also one can argue that due to naturally limited resources, the opponent can only pay the reward up to some finite limit L. When recalculating the expectation value with a finite sum for $N = \log_2 L$

$$\sum_{n=1}^{N} \left(\frac{1}{2}\right)^n \times 2^n \neq \infty$$

one will conclude a quite decent expected award.

So to analyse the expected accumulation length properly, one should not only calculate the expectation value, but rather the full asymptotic distribution of possible accumulation lengths. The distribution is described as asymptotic, as of course it will be a function of the number of steps through the Markov chain.

Let p_{ij} denote the probability for a Markov chain to eventually return to a state $\textcircled{j}$ in finite time T_j^r when starting at state $X_0 = i$.

$$\begin{aligned} p_{ij} &:= \mathbb{P}\left(T_j^r < \infty \middle| X_0 = i\right) \\ &= \mathbb{P}\left(X_n = j \text{ for some } n \geq 1 | X_0 = i\right) \end{aligned} \tag{6.46}$$

According to Proposition 5.1 in (Privault 2018) the probability distribution for the number of returns R_j to a state $\textcircled{j}$ when starting in the state $X_0 = i$ can then be derived as

$$\mathbb{P}\left(R_j = m \middle| X_0 = i\right) = \begin{cases} 1 - p_{ij}\,, & m = 0\,, \\ p_{ij} \times (p_{jj})^{m-1} \times (1 - p_{jj})\,, & m \geq 1\,. \end{cases} \tag{6.47}$$

According to the Equations 5.4.5 and 5.4.6 in (Privault 2018) the expectation value for the number of returns can be written as

$$\begin{aligned} S_{ij} = \mathbb{E}\left[R_j \middle| X_0 = i\right] &= \sum_{m=0}^{\infty} m \mathbb{P}\left(R_j = m \middle| X_0 = i\right) \\ &= (1 - p_{jj})\, p_{ij} \sum_{m=1}^{\infty} m\, (p_{jj})^{m-1} \\ &= (1 - p_{jj})\, p_{ij} \frac{1}{(1 - p_{jj})^2} \end{aligned} \tag{6.48}$$

$$\begin{aligned} S_{ij} &= \frac{p_{ij}}{1 - p_{jj}} \\ S_{jj} &= \frac{p_{jj}}{1 - p_{jj}} \end{aligned} \tag{6.49}$$

by using the differentiation of the geometric series Equation (6.41)

$$\sum_{k=1}^{\infty} k r^{k-1} = \frac{1}{(1-r)^2}\,, \qquad -1 < r < 1\,. \tag{6.50}$$

By solving Equation (6.49) for the return probabilities p_{ij} and p_{jj} one gets

$$\begin{aligned} p_{ij} &= \frac{S_{ij}}{1 + S_{jj}} \\ p_{jj} &= \frac{S_{jj}}{1 + S_{jj}} \end{aligned} \tag{6.51}$$

The values obtained from Equation (6.40) for S_{ij} can now be substituted in order to obtain values from Equation (6.51) for another substitution in Equation (6.47). This then gives the distribution for the number of returns to state $\textcircled{j}$, starting in state $\textcircled{i}$. Again, the desired distribution for the

number of accumulated events is obtained when evaluating this for $i = 0$ at $\textcircled{0}$ and $j = 1$ at $\textcircled{k_{++}}$.

6.2.4 Accumulation Time

In the previous Section, the distribution and expectation of the possible accumulation lengths have been analysed. The focus shall now lie on the time, it takes to accumulate that number of events until the packet is closed by a conflict. This will help to evaluate, how long the events can be buffered in the accumulation process and whether the condition C.2 on page 77 will have a significant effect or not. Hereby one can then define a constraint for the value of the timeout: The timeout should be chosen such that it is slightly higher than the expected accumulation time. This way, the timeout can limit the accumulation, in case a low event rate hinders full accumulation. Usually the timeout should not come to significant effect, as the packet will mostly be closed by conflict rather than the configured timeout.
In contrast to the accumulation length, the accumulation time will also depend on the input event rate which has to be incorporated in the transition probabilities in Equation (6.18) and Equation (6.22) of the Markov chain model.

6.2.4.1 Mean Accumulation Time

The mean accumulation time, can be identified as the expected number of steps through the graph in Figure 6.3 until reaching the final absorbing state $\textcircled{\hat{k}}$. In (Privault 2018) this is called the *First Hitting Time T_m* for the only absorbing state $\textcircled{m}$ when starting in state $\textcircled{i}$. It can be derived from Equation (6.40) by adding up the expected number of returns for all other states:

$$\mathbb{E}\left[T_m \mid X_0 = i\right] = 1 + \sum_{\substack{j \in \mathbb{S} \\ j \neq m}} S_{ij}\,, \qquad i \neq m \tag{6.52}$$

6.2.4.2 Distribution of Accumulation Times

The distribution of the absorption time in a finite discrete time Markov chain with a single absorbing state is also referred to as discrete phase-type distribution. According to Equation 1.11 in (Nielsen 2020) the event of absorption after n steps through the chain can be partitioned into two sub events. First, the process still has to be in a transient state after $(n-1)$ steps and second the absorption shall happen exactly after the n'th step. Therefore, using Equation (6.17) the probability for absorption in state $\textcircled{\text{m}}$ after n steps can be written as

$$\mathbb{P}\left[X_n = m | X_0 = i\right] = \left[\left(P_T\right)^{n-1} P_A\right]_i \tag{6.53}$$

where P_T and P_A are defined through Equation (6.42). P_T describes the transient states transition matrix while P_A describes the transition matrix into the absorbing states.

6.2.4.3 Accumulation Time and Accumulation Length

A possibly more simple method to get a value for the expected accumulation time is to derive it from the previously obtained expected accumulation length (cf. Equation (5.5)). For this, one needs

to know the mean rate of events that are statically assigned to the respective bucket under analysis. The mean ISI derived from the mean event rate at the bucket input can then simply be multiplied to the expected accumulation length. The difference to the previous approach of modifying the transition probabilities (cf. Section 6.2.2.6) and re-evaluating the Markov Chain is that there the rate-probability would be determined for all events coming from the neuromorphic chip, regardless whether they are assigned to our particular bucket or not. The assignment constraint is rather applied later, in the Markov analysis itself through the particular transition probabilities.
By combining this approach to to accumulation time with Equation (5.6) or Equation (5.7) while demanding, that the time until conflict shall be larger than the accumulation time required for reducing the header overhead below a certain threshold, the event rate cancels out and one arrives at a requirement constraint for the expected accumulation length:

$$\begin{aligned} & \frac{\mathbb{E}\left[N_{\mathrm{acc}}\right]}{r_{\mathrm{ev}}^{d^\star}} \geq l_{\mathrm{acc}} \geq \frac{s_{\mathrm{h}}}{s_{\mathrm{ev}} \cdot r_{\mathrm{ev}}^{d^\star}} \cdot \frac{e_{\mathrm{pl}}}{o_{\mathrm{h}}} \\ \Leftrightarrow \quad & \mathbb{E}\left[N_{\mathrm{acc}}\right] \geq \frac{s_{\mathrm{h}}}{s_{\mathrm{ev}}} \cdot \frac{e_{\mathrm{pl}}}{o_{\mathrm{h}}} \end{aligned} \tag{6.54}$$

6.3 Results of the Mathematical Analysis

The Markov chain analysis described above shall now be used to evaluate the number of accumulatable events in a bucket under the ideal condition that only destination conflicts (Condition C.3) are of relevance. As stated in Section 6.2.2, this analysis is in general only possible for static assignments. Therefore, the Modulo assignment strategy (AS.1) is exemplarily regarded here on top of different example distributions of event destinations in Sections 6.3.2 to 6.3.4. The uniform, normal and triangular destination distributions as defined in Section 6.3.1 will here exemplarily be examined. It should be noted that these distributions are not claimed to be of any particular meaningfulness with regard to real neural network models.
Finally, a linking relationship will be found between the assignment strategy and the resulting expected number of accumulated events in Section 6.3.5. This relationship will allow, to define an optimisation strategy for a static assignment table (AS.2) in Section 6.3.6.

6.3.1 Event Destination Distributions

Before beginning with the actual analysis, first some basic exemplary distributions used to test the behaviour of the accumulation process will be introduced. Each event presented at the input of the accumulation system is labelled with a destination address. The probability for a particular destination to be requested with the current event (referenced as $P(e)$ in Section 6.2.2) is distributed across all available destinations. In the following analysis this probability distribution will be assumed to one of the following cases:

DD.1 Uniform Distribution: Each destination has the same probability of occurrence

$$P(d) = \frac{1}{D} \tag{6.55}$$

where D is the total number of destinations.

DD.2 Normal Distribution: This is the standard normal distribution with a normalised Gauss curve

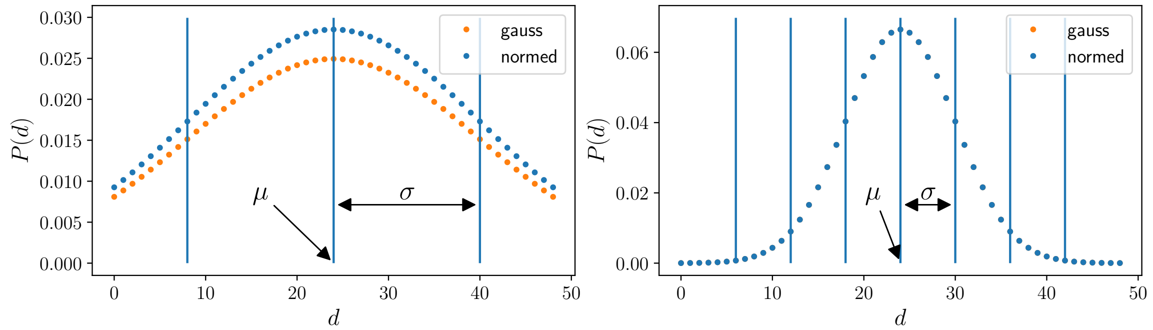
$$P(d) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{d-\mu}{\sigma}\right)^2} . \tag{6.56}$$

with a standard deviation σ around a mean value μ. The left part of Figure 6.4a shows an exemplary plot of a discrete normal distribution. As this analysis is dealing with discrete distributions on a constrained interval, the distribution probabilities have to be renormalised to sum up to 1. The effect of this renormalisation can be seen in the left plot of Figure 6.4a. The destinations d are here distributed with a relatively broad gauss curve leading to clipping the boundaries of the curve. Because of this, the renormalised probabilities are higher than the pure gauss curve to compensate for the clipped boundaries. However, in the right plot in Figure 6.4a which shows a relatively narrow gauss curve, the renormalisation does not lead to a visible difference and both curves overlap. This is because the clipped boundaries are not a significant loss to the overall probability sum, as already 99.8 % of the probability mass are inside a 3 σ interval of the normal distribution.

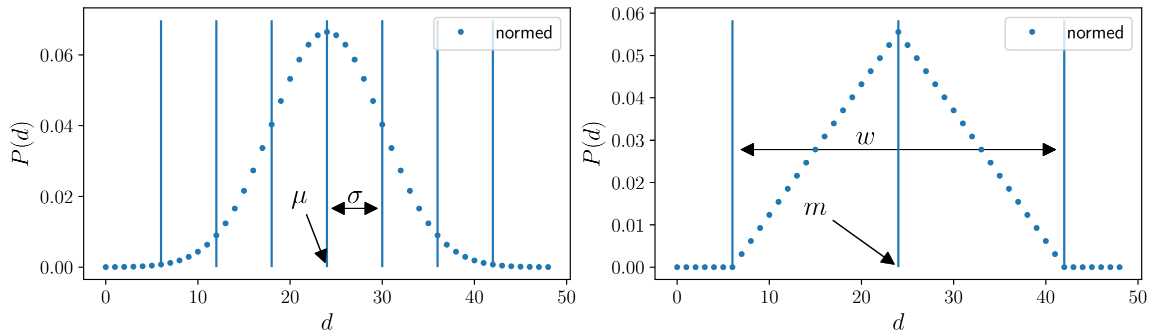
DD.3 Triangle Distribution: This distribution has a triangular shape with a given baseline width w around a middle value m. The probabilities rise and fall linearly before and after the centre position. Outside the boundaries defined by the w parameter, the probabilities are clamped to a very small value (≈ 0). They are not clamped to exactly 0, as this would lead to singular matrices in Equation (6.44). The right plot in Figure 6.4b shows an exemplary triangular distribution and opposes it to a normal distribution of comparable width (left plot). To have the triangle distribution comparable in width to a normal distribution, one can assume $w = 6\sigma$, as the normal distributed probabilities are near 0 outside this 3 σ interval.

$$P(d) = \begin{cases} \approx 0 & d < m - \frac{w}{2} \\ \frac{4}{w^2}(x-m) + \frac{2}{w} & d \geq m - \frac{w}{2} \ \wedge \ d < m \\ -\frac{4}{w^2}(x-m) + \frac{2}{w} & d \geq m \ \wedge \ d \leq m + \frac{w}{2} \\ \approx 0 & d > m + \frac{w}{2} \end{cases} \tag{6.57}$$

Of course non of these distributions is expected to resemble a real distribution occurring with a real neural network model, but purely intended as benchmark cases for the accumulation system. The real distribution of occurring destinations on the interconnection network, as modelled here, depends on several factors. First of all, it clearly depends on the connectivity structure and activity of the modelled neural network. However, it is strongly influenced by how this neural structure is mapped onto the system, i.e. which neural populations are or are not placed together on the system nodes (cf. Figure 5.2 and Figure 6.1). In Figure 6.1, the resulting distribution of destination nodes is explicitly indicated at the node edges. If the neural populations were placed differently onto the system units, other projections would become relevant, as crossing the chip boundaries. Generally, it would seem a good strategy to place frequently communicating populations together on a common chip and distribute these population clusters across the system, in a way such that frequent common



(a) Comparison of a discrete normal distribution of 1.5 σ radius (**left**) and one with 4 σ radius (**right**). The orange curve shows the un-normalised discrete gauss-curve.



(b) Comparison of a discrete normal distribution (**left**) and a discrete triangle distribution (**right**).

Figure 6.4: Exemplary plots of the used discrete destination distributions. The parameters of the distributions are annotated in the respective plots.

communication partners are gathered together.

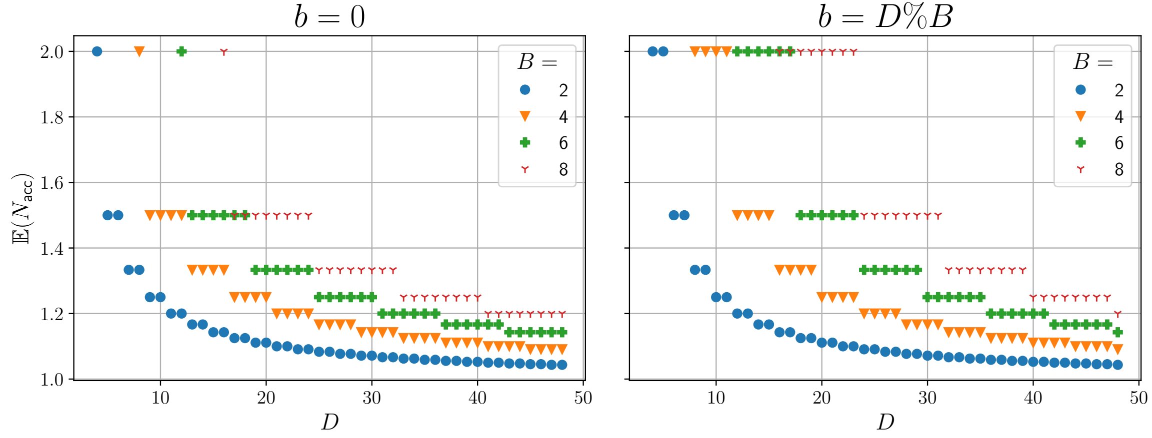
In the following sections, the term *distribution* refers to one of these distributions DD.1 to DD.3, describing the probability of a specific destination occurring with an input event.

6.3.2 Uniform distribution

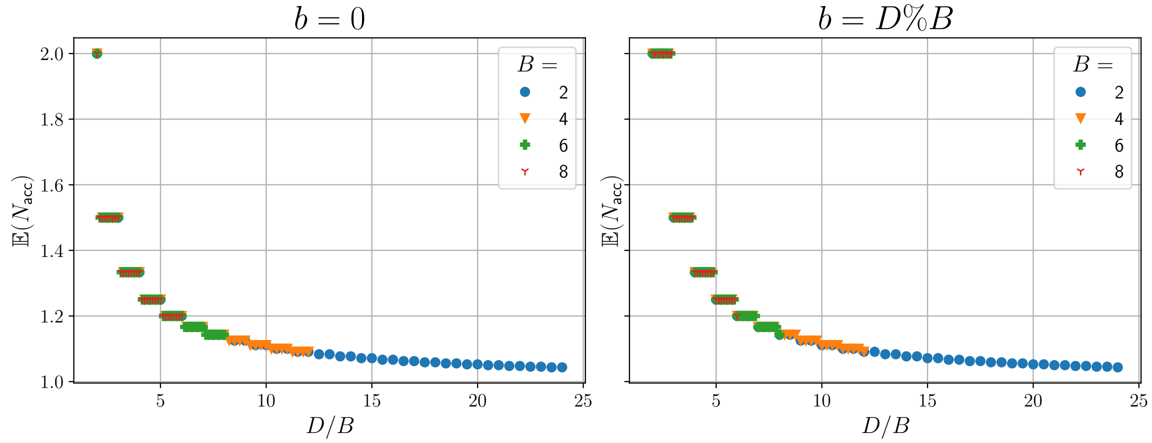
By using Equation (6.44) one can compute the expected number of accumulated events $\mathbb{E}(N_{\text{acc}})$ in a bucket b in a collection of buckets B with events whose destinations are uniformly distributed. The event destinations are assigned to the collection of buckets using the modulo strategy described in AS.1 on page 79. This calculation is repeated, while sweeping the number of destinations D in a range of $[2B, 48]$ with different values of B. The expected filling level $\mathbb{E}(N_{\text{acc}})$ is computed for each bucket b in the respective collection. Figure 6.5a shows the result of this analysis. The resulting curves, each displaying the result for a different number of buckets B, show a stepped pattern originating from the Modulo assignment strategy. As each destination d, added to the set of destinations, is assigned to the next bucket in the collection, a particular bucket b is only assigned an additional destination after B destinations have been added to the system. Only then, the situation for that bucket will change, resulting in a lower expected accumulation length.

As to Equation (6.8), all buckets with IDs $b < D\%B$ will have one more assigned destination than buckets with IDs $b \geq D\%B$. Therefore the expectation values on the right plot do equal the next higher expectations on the left plot.

Figure 6.5b shows the same plots as a, but now the X-axis shows the relative number of destinations



(a) Plot over the absolute number of destinations D for different numbers of Buckets B.



(b) Plot over the relative number of destinations D per number of Buckets B (D/B) for different numbers of Buckets B.

Figure 6.5: Plots of the expected accumulation length $\mathbb{E}(N_{\text{acc}})$ for uniform-distributed destinations which are modulo-assigned to the available buckets b. **left:** Plot for the bucket with ID $b = 0$. **right:** Plot for the bucket with ID $b = D\%B$.

per bucket D/B. From this plot one can see that the expected accumulation length only depends on this relative quantity, as all previously scattered curves now fall together to a single stepped line. It also shows that with a lower number of buckets this curve is sampled on a greater interval reaching to larger ratios and therefore lower expectations. The maximum occurring expectation equals two for a relative number of two destinations per bucket with $b < D\%B$ or the equivalent with $b \geq D\%B$. This can be explained by the fact that in this case each bucket is assigned two destinations and as every destination is equally probable within the uniform distribution, the accumulation process is interrupted on average at the second incoming event after the first one has been accepted.

Figure 6.6 shows the probability distribution for the resulting number of accumulated events in a particular bucket in the collection, calculated through Equation (6.47). It can be seen that this distribution is also independent of the number of buckets and only depends on the relative number of destinations per bucket D/B which is encoded in the colourmap in the plot. The most probable case is always the accumulation of a single event, as the bucket will in any case accumulate at least one event before a conflict-condition can occur. For lower ratios D/B the distribution and expectation

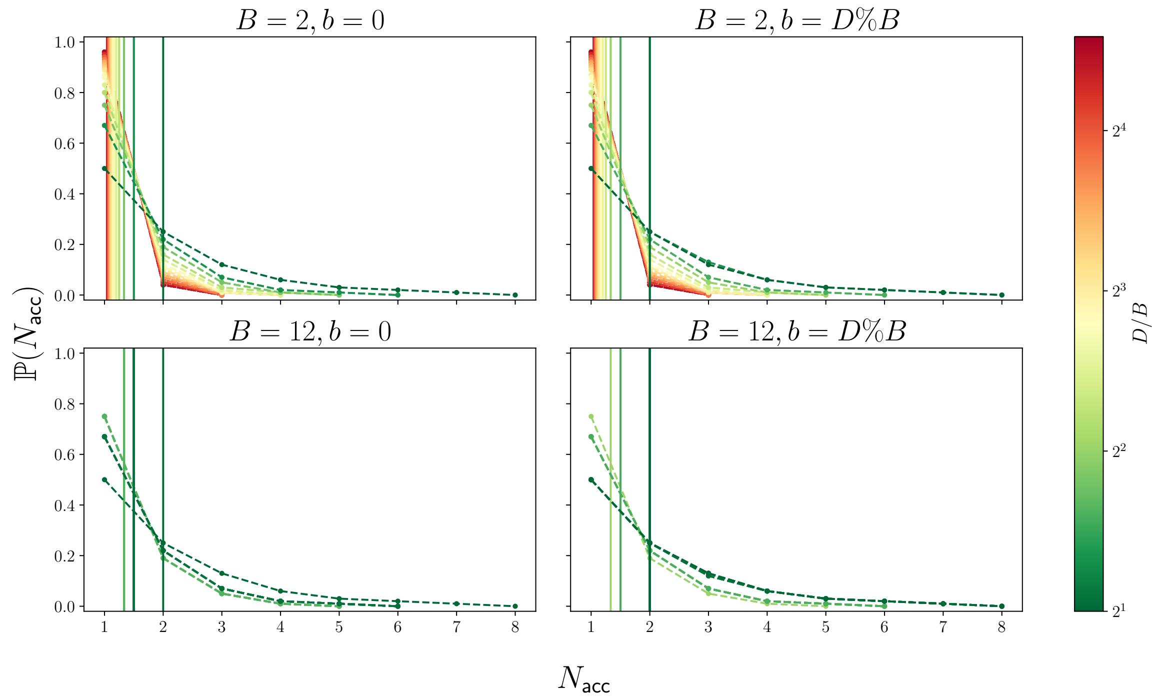


Figure 6.6: Plots of the probability distribution $\mathbb{P}(N_{acc})$ for the accumulation length for uniformly distributed destinations. The relative number of destinations per bucket D/B is imprinted on the distribution curves using a colour code. Green encodes low ratios while red encodes high ratios. The scale of the colourmap is chosen as base-2-logarithmic in order to have a better contrast in the area where the most curves gather. The vertical lines show the expectation value belonging to the distribution plotted with the same colour.

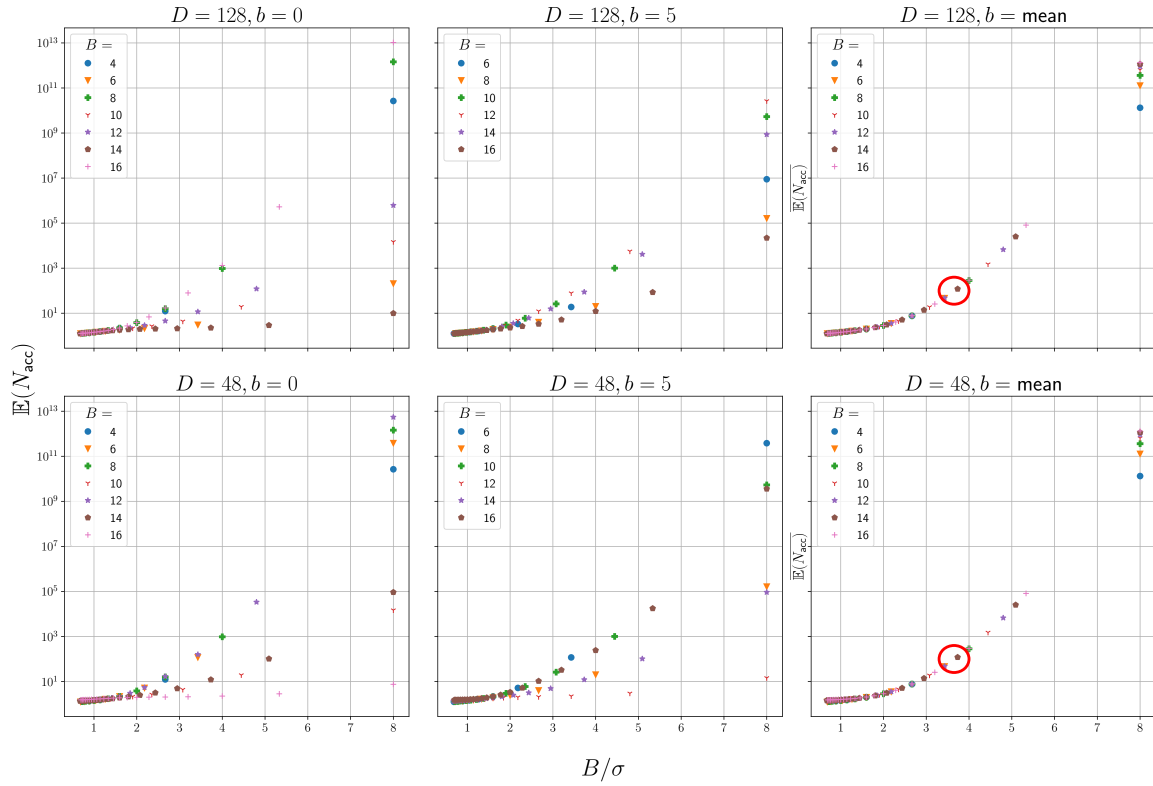
value shift right towards higher accumulation lengths. By looking closely, one can again find the shift in expectation between the buckets $b < D\%B$ and $b \geq D\%B$ (cf. the left plots to the right ones). For high numbers B, there are less ratios D/B sampled by the analysis compared to low ratios, which can be seen in the comparison of the number of curves in the plots on the upper and lower half of Figure 6.6.

From this plot one can read that even for the best ratios plotted in dark-green, the probability for accumulating more than 4 events is less than 10 %.

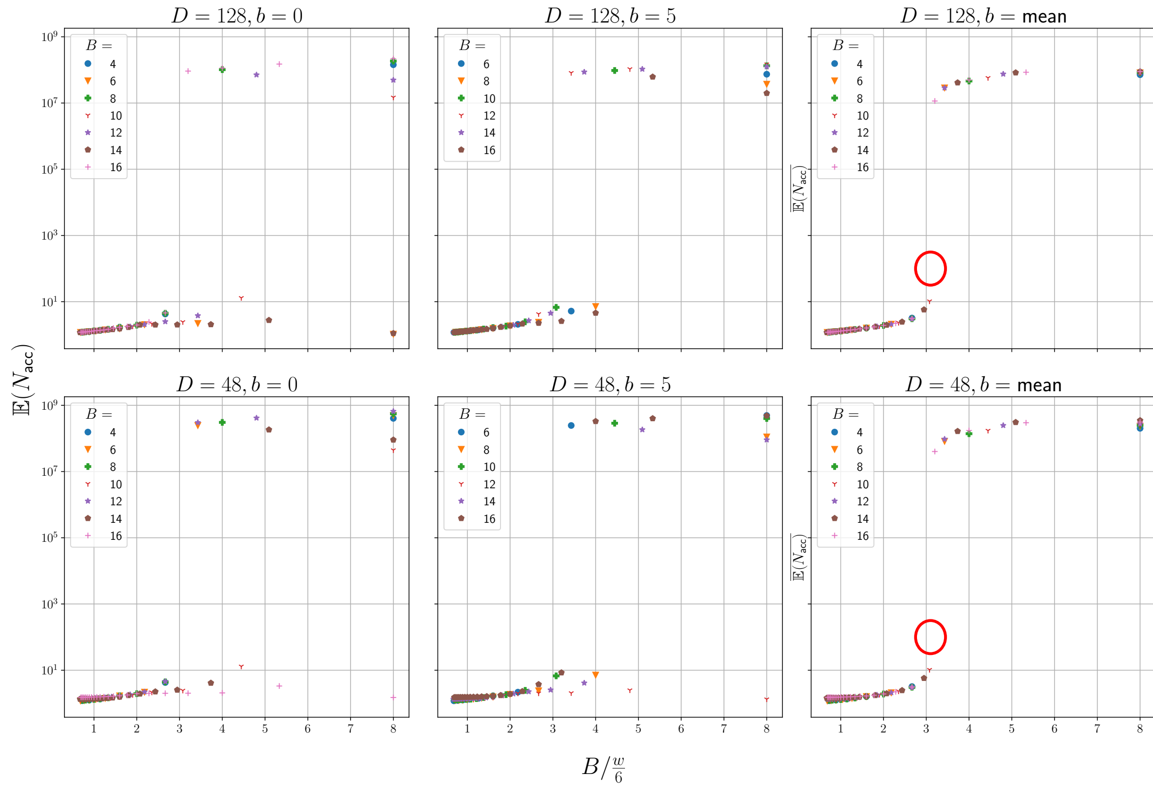
6.3.3 Normal and Triangle Distribution – Scaling the Width

Now, the accumulation process shall be analysed for events with normal and triangle distributed destinations. The destinations are again assigned to the collection of B buckets by modulo operation (AS.1 on page 79). For a given number of overall destinations D the σ- and w-width of the distribution is scaled in a range between $\left[\frac{1}{8}B, 1.5B\right]$ and the expected number of accumulated events is again calculated for each bucket b in the collection using Equation (6.44). Figure 6.7 plots the acquired data across the relative number of buckets per unit σ- or w-width B/σ or $B/\frac{w}{6}$ of the respective distribution. There, Subfigure a shows the results for the normal distribution while Subfigure b shows the results for the triangle distribution.

From both Figures it can be seen that the individual curves' slopes vary quite randomly across the number of buckets B and destinations D, as well as for the individual bucket b (left and middle



(a) Normal Distribution.



(b) Triangle Distribution.

Figure 6.7: Plots of the expected accumulation lengths $\mathbb{E}(N_{acc})$ for different numbers D of randomly distributed destinations in different buckets b (**left** and **middle**). The X-axis shows the number of buckets available per unit width of the respective distribution. The two plots on the **right** show the mean expected accumulation length $\overline{\mathbb{E}(N_{acc})}$ averaged over all available buckets in the collection.

columns). However, when averaged over all buckets b in the respective collection (right columns), the mean expectation does much less depend on these parameters, but mostly on the ratio of buckets per unit σ or $\frac{w}{6}$ of the respective distribution, shown on the X-axis. Only for large ratios around 8 buckets per σ a significant difference in the mean expectation value is observable between different numbers of buckets B for the normal distributed destinations in Figure 6.7a. However, these apparently still do not depend on the number of destinations D. The missing dependency on the absolute number of destinations can be explained by the shape of the distributions, as the destinations outside the σ or w width do not significantly contribute due to their vanishing probability. The reason for the curves' slopes' random behaviour in the left and middle columns is believed to be an artifact of the Modulo assignment strategy, as the destination ids assigned to an individual bucket change while the overall numbers change. This is a similar effect as that observed in Figure 6.5 between the left and the right, but now it looks more complex, as the destination distribution is not uniform anymore. These effects are neutralised, when averaging over the different bucket instances.

The reason for the $\frac{w}{6}$ scaling on the X-axis in Figure 6.7b is explained in Section 6.3.1 and ensures the comparability of the two distributions' widths. From the plots, one does notice that with triangle distributed destinations, there seem to be a phase-transition between very low expected accumulation lengths at $O(5)$ and very high ones at $\approx 10^8$. In contrast, with normal distributed destinations the expected accumulation length rises without an infliction point on the curve.

Evidence for the absence of an infliction point with normal distributed destinations (also for higher B/σ and $\mathbb{E}(N_{\text{acc}})$) can be found in the comparison of the spreading shape with the different parametrisations of the two distributions. While in Figure 6.7b the horizontal position of the phase change varies with the parametrisation, Figure 6.7a only varies the rising-slope of the curves manifesting in the vertical spread of points on the right end of the X-axis.

From Figure 6.7 one can read (the **red ellipses**) that on average at least around 3.6 buckets per unit σ or 3.1 buckets per unit $\frac{w}{6}$ are needed to achieve an expected accumulation of $\approx 10^2$ events, corresponding to a full EXTOLL packet (cf. Appendix B.2), until a conflict arises.

6.3.4 Normal and Triangle Distribution – Scaling the Destinations

In the last Section, the distribution width of a given number of destinations has been swept for each number of buckets B. Here, the effect of sweeping the number of destinations for each number of buckets with a given relative width of the distribution will be analysed. Figure 6.8 shows the result of this analysis for different relative distribution widths.

Again for having the normal and the triangle distribution comparable, triangle distribution width is scaled with a factor of $\frac{1}{6}$, as the normal distribution will include most of its probability mass in a $3\,\sigma$ interval, as described in DD.2 and DD.3 on page 92.

The shape of the curves in Figure 6.8 looks much like in Figure 6.7. This is expected, as the X-axis scale is directly connected between both figures through a linear factor. Through this linear factor, the scaled number of overall destinations also automatically scales the absolute width of the distribution.

It should be noted that each data curve in Figure 6.8 is a cumulation of the data from the different numbers of buckets B. This confirms the previous result that the plotted expected accumulation length is on average independent of the number of buckets and instead mostly depends on the relative

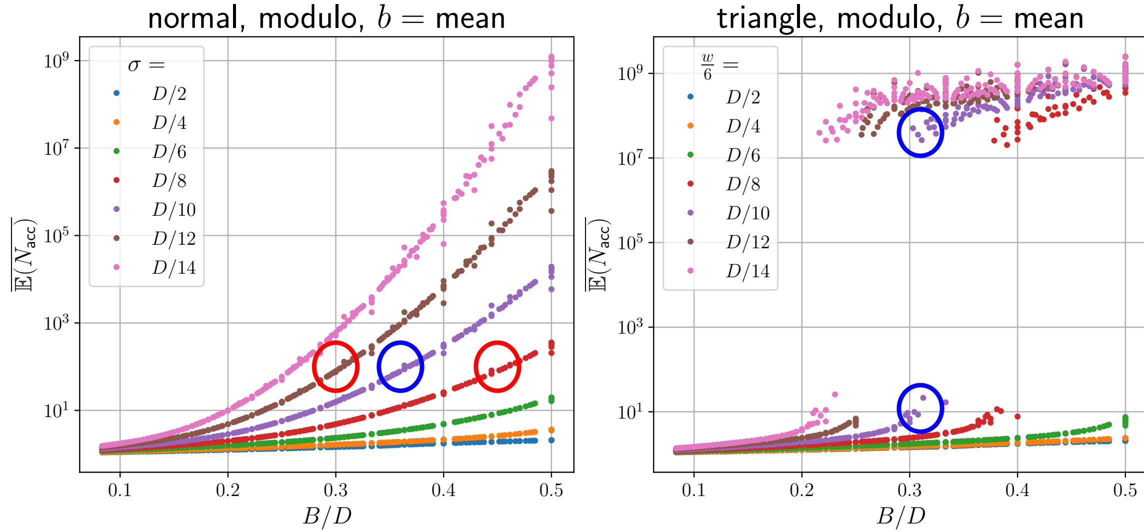


Figure 6.8: Plots of the mean expected accumulation length $\overline{\mathbb{E}(N_{\text{acc}})}$ cumulating data from different numbers of buckets for different distribution widths. **Left:** Normal distribution with different relative σ-widths. **Right:** Triangle distribution with different relative $\frac{w}{6}$-widths.

number of buckets per distribution width.

Also, from Figure 6.8 one can now see (the **red ellipses** in the Figure) that having a more narrow distribution with respect to the number of destinations generally leads to a higher expected accumulation length. For example when looking at the left plot, one can read that with a σ-width of $\frac{D}{8}$ at least 0.45 buckets per number of destinations D are needed to accumulate more than ≈ 100 events, while with a σ-width of $\frac{D}{12}$ only 0.3 buckets per number of destinations D are required for the same expected accumulation length.

By comparing the normal distribution on the left with the triangular distribution on the right, one can also see that the latter reaches the goal of ≈ 100 expectedly accumulated events a little sooner than the former. For example with the triangle distribution and a width of $\frac{D}{10}$ only 0.31 buckets per number of destinations D are required to reach the phase transition between low and high accumulation lengths at $\approx 10^8$ events before conflict while with the normal distribution at least 0.36 bucket per number of destinations D are needed to gradually exceed an expectation of ≈ 100 events before conflict (cf. **blue ellipses** in the Figure).

6.3.5 Approximating the Expected Accumulation Length

Up to now, only the rather naive Modulo assignment strategy (AS.1 on page 79) has been applied to the accumulation process. However, as stated in Section 6.1.2.3 there should be a way to optimise the assignment for a minimum of conflicts, using a freely programmable assignment table (AS.2). In order to arrive at such an optimised assignment strategy, first an intuitive approximation for the expected filling level is developed. Having that, two optimisation approaches will be compared to respect to the developed approximation measure.

Intuitively, the expected number of accumulated events in an individual bucket b corresponds to the frequentness of conflicts with regard to the currently accumulating destination and the destination of the current event at the input. As explained in Section 6.2, this largely depends on the individual

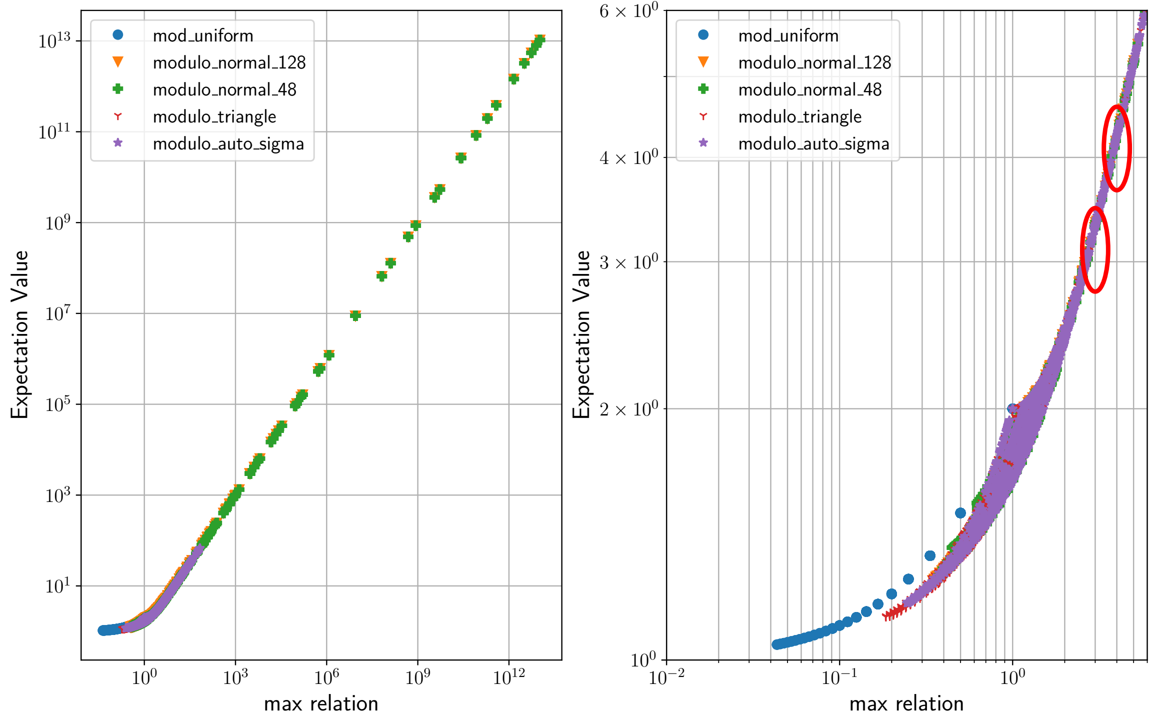


Figure 6.9: Plot of the expected accumulation lengths against the ratio defined in Equation (6.58). The data is a collection from the different previous analyses. All data points are generated with Modulo assignment, but different distributions. `mod_uniform`: uniform distribution. `modulo_normal_128`, `modulo_normal_48`: normal distribution with 128 and 48 destinations respectively. `modulo_triangle`: triangle distribution. `modulo_auto_sigma`: a normal distribution, scaled such that all destinations fit in a $3\,\sigma$ interval. **Left:** Full scale plot. **Right:** Detail zoom into the lower left part of the plot.

destinations assigned to that bucket, as well as their relative probabilities with respect to that bucket. The can now be approximated by regarding the amount of disturbance of the most probable destination's accumulation by all other destinations assigned to the particular bucket. This can be expressed by the ratio of the most probable destination's probability to the sum of all other destinations' probabilities at that bucket.

$$\mathbb{E}(N_{\text{acc}}) \approx \frac{P_b(d)}{\sum_{d' \neq d, P(d' \to b) \neq 0} P_b(d')} \tag{6.58}$$

In order to obtain the probability of occurrence to a bucket for each destination assigned to that bucket, one has to normalise the individual probabilities with respect to the sum of all destinations assigned to that bucket.

$$P_b(d) = \frac{P(d)}{\sum_{d', P(d' \to b) \neq 0} P(d')} \tag{6.59}$$

Figure 6.2 shows these normalised probabilities with respect to the respective buckets.
To confirm this educated guess, Figure 6.9 shows a plot of the expected accumulation length against this approximation ratio. The right side of the Figure shows a zoom-in on the lower left part of the plot. From this plot one can read that the proposed approximation is very good for ratios down to around five.

For lower ratios, the deviation from the linear curve slowly increases until the ratio gets down to around three and then starts to become quite significant (cf. **red ellipses** in the Figure). For low ratios, the expected accumulation length is higher than approximated. This is expected, as the accumulation length cannot become lower than one and the expectation value will therefore converge to one for ratios approaching zero. Intuitively this behaviour can also be explained by stating that for low probability ratios, the other destinations assigned to the bucket become more and more important compared to the most probable destination. Therefore the most probable destination does have a less significant advantage over the other destinations and a higher proportion of time is spent (not) accumulating other destinations.
Given this intuitive approximation one can now also argue that a uniform distribution, as analysed in Section 6.3.2 is the worst case distribution, as there is no distinguished most probable destination and the ratio for any destination selected as the enumerator will be near zero. This is also directly visible in the right plot of Figure 6.9, as the data-points belonging to the analysis in Section 6.3.2 (`mod_uniform`) are plotted at very low ratios as expected.

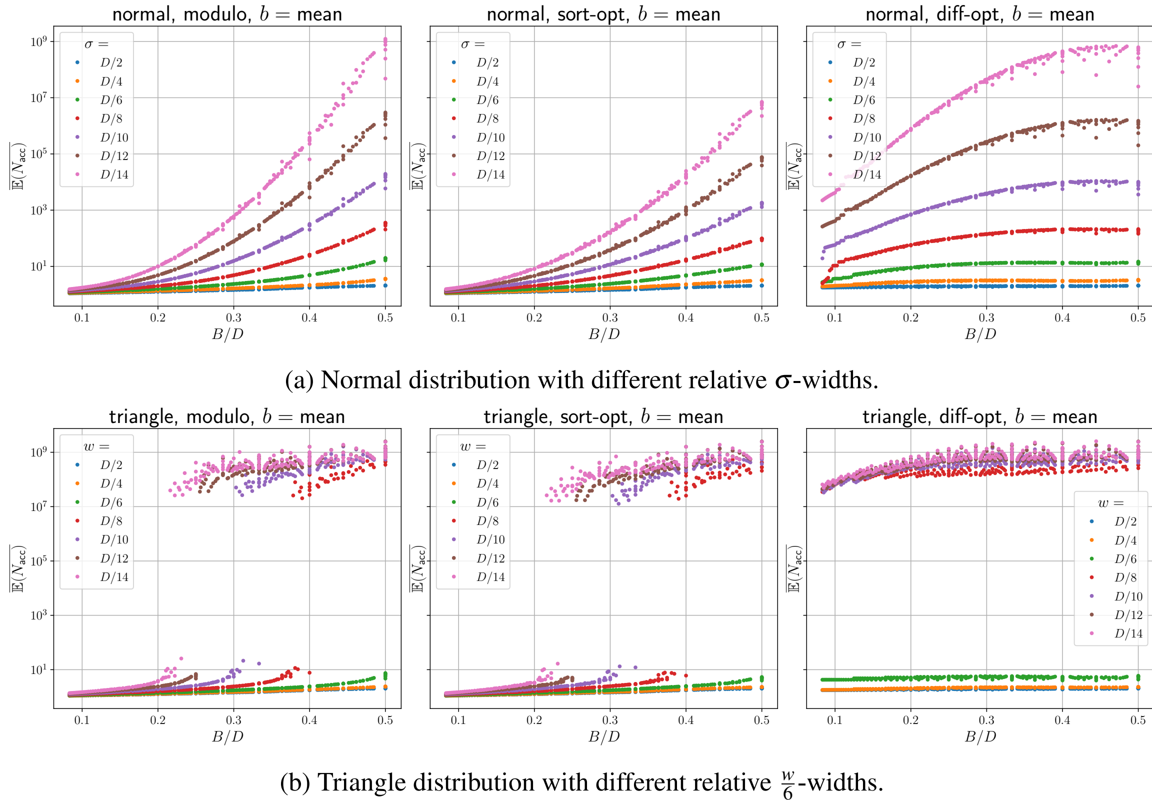
6.3.6 Optimising the Expected Accumulation Length

In the following, two custom assignment strategies for table-based assignment (AS.2 on page 79) will be defined under the objective to optimise the number of accumulated events per bucket. Both strategies aim to maximise the previously motivated ratio between the most probable destination's probability and the sum of all other destinations' probabilities for all buckets.

OS.1 Sort-Opt: This strategy first sorts the list of available destinations with respect to decreasing probability of occurrence. Then it does a Modulo assignment of the destinations, following the previously sorted list. Thereby the hope is that the most probable destinations are not assigned to the same bucket as similarly probable events.

OS.2 Diff-Opt: With this strategy, the destinations are iteratively assigned to buckets. For each assignment, first an intermediate probability ratio is calculated for each bucket; once before the assignment and once with the destination hypothetically assigned to the respective bucket. The destination is then finally assigned to that bucket where the signed ratio-change is maximal, i.e. either increases most or decreases least. As the ratios are not defined at the first assignment step, the first round of destinations is assigned using a modulo operation.

Figure 6.10 shows a comparison between these Optimisation Strategies OS.1 to OS.2 using the analysis described in Section 6.3.4 and Figure 6.8. It can be seen that the *Sort-Opt* strategy does not have any significant effect on the triangle distribution and even worsens the accumulation with the normal distribution. However, the *Sort-Opt* strategy greatly improves the expected accumulation length for all numbers of buckets, for the normal distribution, as well as for the triangle distribution. With the triangle distribution, the improvement is even better and completely removes the previously observed phase transition between low and high accumulation lengths. Instead, for each analysed distribution width it unifies the expected mean accumulation length to the respective maximum expected accumulation length observed in the unoptimised analysis.
One can try to understand the poor performance of the *Sort-Opt* strategy when looking at the result of the sorting operation in Figure 6.11. As the two distributions used here already follow a smooth



(a) Normal distribution with different relative σ-widths.

(b) Triangle distribution with different relative $\frac{w}{6}$-widths.

Figure 6.10: Plot the same analysis as in Figure 6.8, but with the two optimisation strategies in comparison. **Left:** The original analysis. **Middle:** Optimisation using OS.1. **Right:** Optimisation using OS.2.

course they partition into two sorted regions. Also, each probability value occurs twice in the respective distribution. Therefore, by sorting the distribution, the distance of different probability values, modulo-assigned to a particular bucket, will approximately be halved. This effect corresponds to the visible reduction in the sorted distribution slope compared to the original distribution slope.

The good performance of the *Diff-Opt* strategy is also quite understandable, as it follows a gradient ascent algorithm. However, this optimisation strategy leads to a highly asymmetric assignment in that form that apparently every bucket is assigned one further destination beyond the initial modulo round and all other destinations are assigned to a single bucket. This effectively leads to all but one buckets having only two destinations assigned and therefore performing quite well and a single bucket performing horribly and being interrupted very frequently. This also explains, why the expected mean accumulation length does less depend on the number of buckets available per number of destinations as observed in Figure 6.10. This *bad-bucket* effect has to be addressed for further optimisation, as it will certainly have a negative impact on the overall performance, as it produces a high packet rate with large overhead.

The further improvement of the optimisation algorithm is not part of this thesis and will be left for future work.

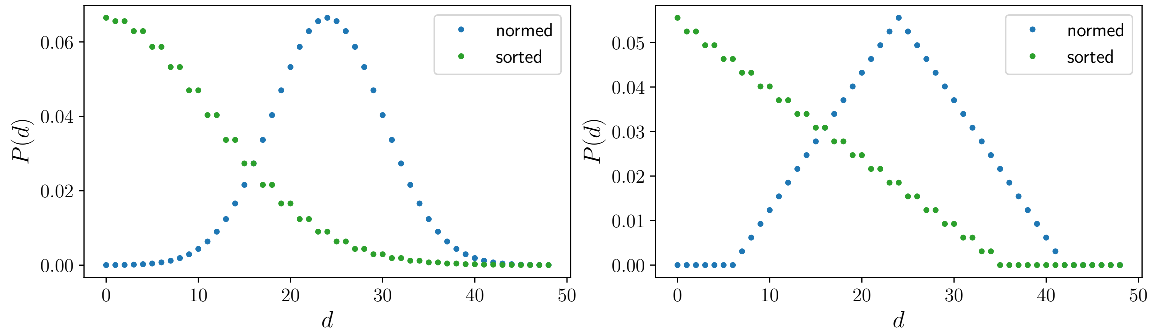


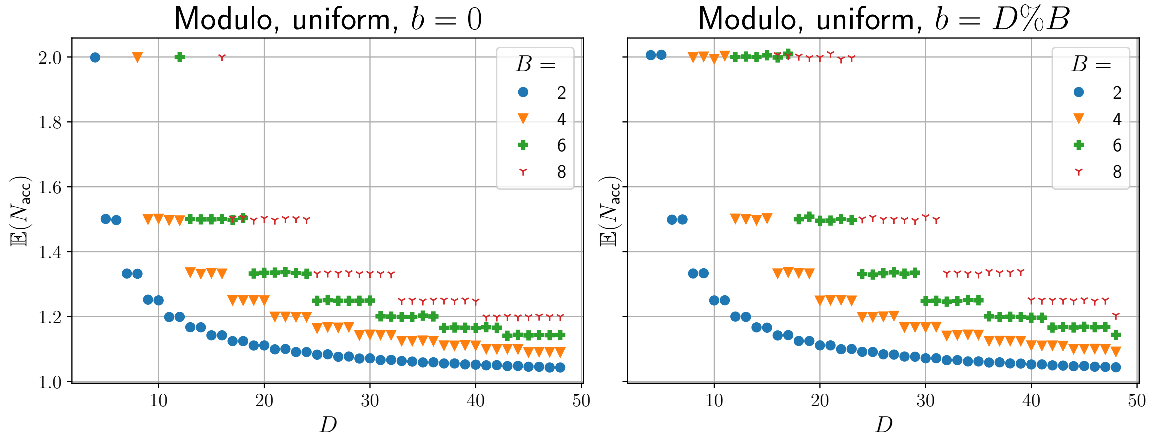
Figure 6.11: Normal (**left**) and Triangle distribution (**right**) sorted with descending probabilities.

6.4 Simulation Analysis

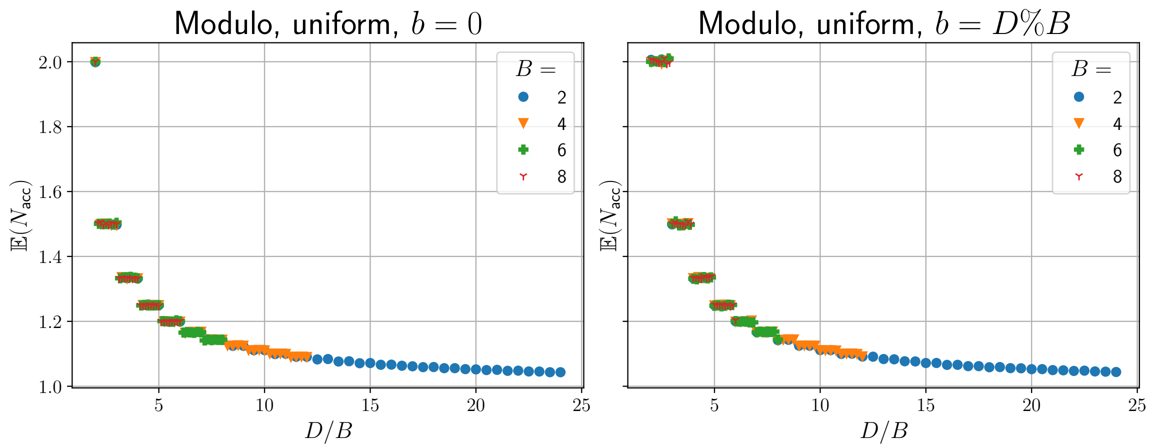
Up to now only static assignment strategies have been analysed. As described in Section 6.2.2.5, dynamic assignment strategies cannot be analysed with the Markov Chain model, as they introduce a dependency on the current time step leading to the invalidity of Equation (6.17).
In order to yet analyse the characteristics of dynamic assignment strategies one must directly model the accumulation system and simulate the dynamic behaviour. Of course there is no reason to not also evaluate static assignment strategies using such a simulation. However, one should notice that the computing time of the simulation is much higher than that of a simple numerical calculation of the desired quantity.
In each step the simulation will create an event with a destination drawn from a constrained random generator. The random generator is constrained to obey a specific distribution as for example a uniform or normal distribution. The simulation then tries to insert the generated event into the bucket of interest. Thereby it is checked, whether the assignment condition is met or whether the event's destination belongs to another bucket and can therefore be ignored. When a conflict occurs, the accumulation length is recorded and the assignment condition is updated. Whether there is a conflict or not, is determined according to the respectively modelled assignment strategy.

6.4.1 Simulating Modulo Assignment

The results of simulating the modulo-assignment will now be compared to the previously presented results of the mathematical Markov Chain analysis of the same assignment strategy and destination distributions.
Figure 6.12 shows the results of a simulation with uniformly distributed event destinations and Modulo assignment (AS.1). When comparing this to the results obtained from the Markov Chain analysis in Section 6.3.2, shown in Figure 6.5, a great accordance between the two methods is found. This verifies both the correctness of the employed mathematical model and the simulation. The only observed difference is a slight random variation in the simulated accumulation lengths that is most clearly visible in Figure 6.12a. This variation arises from the random number generator, used in the simulation in contrast to the mathematical analysis that does not employ a random generator but rather deterministically calculates the expectation values based on fixed probabilities.
Repeating this comparison between simulation and mathematical analysis for normal distributed event destinations leads to the results shown in Figure 6.13. The result of the Markov Chain analysis



(a) Plot of the expected accumulation length over the absolute number of destinations D for different numbers of buckets B

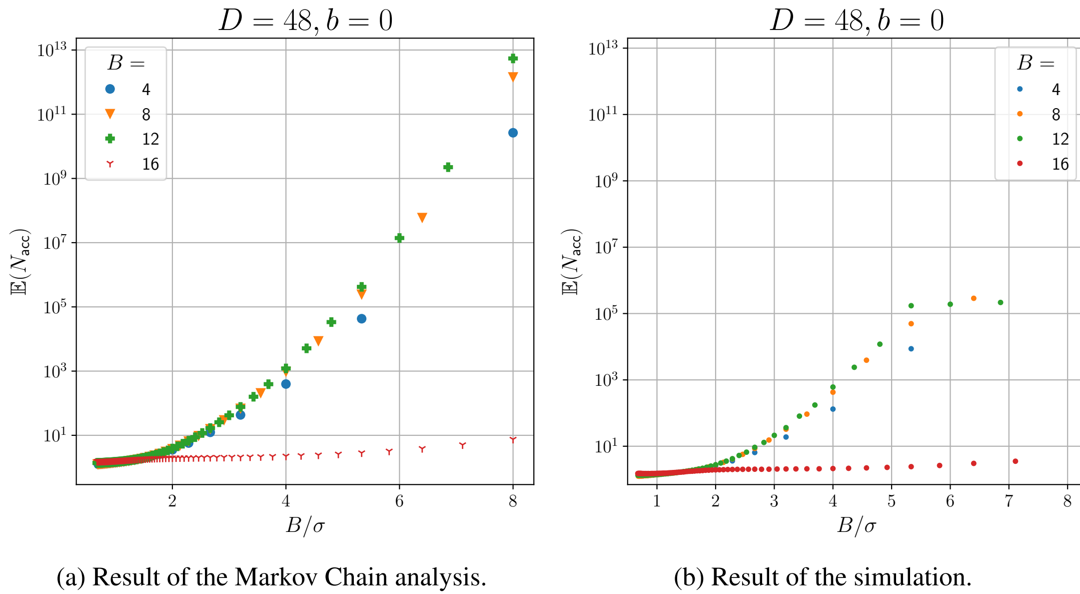


(b) Plot of the expected accumulation length over the relative number of destinations per number of buckets D/B for different numbers of buckets B

Figure 6.12: Plots of the expected accumulation length $\mathbb{E}(N_{acc})$ as obtained by the simulation for uniform-distributed destinations which are modulo-assigned to the available buckets b. **Left:** Plot for the bucket with ID $b = 0$. **Right:** Plot for the bucket with ID $b = D\%B$.

shown in Figure 6.13a has already been presented in the bottom left pane of Figure 6.7a and is repeated here for means of more easy comparison. When comparing the simulation results to the numerical results, the corresponding curves pose a good match for low ratios B/σ ($B = 4$, 8 and 12 buckets). For high values B/σ (above ≈ 5), where the expected accumulation length exceeds 10^5 in Figure 6.13a, the simulation result in Figure 6.13b saturates while the numerically produced curves further diverge. The reason for this behaviour is hypothesised to be the limited number of simulation cycles. As each simulation cycle handles a single event, the simulated accumulation length is rigorously limited to the number of cycles executed. This hypothesis agrees with the observation that the curves in Figure 6.13b do not exceed the number of simulation cycles, which was 10^6 in the creation of this Figure. The hypothesis is also supported by the observation that the curve created for $B = 16$ buckets which does not exceed this limit in Figure 6.13a is also not distorted in Figure 6.13b.

This hypothesis is also verified by repeating the simulation with the same parameters, but now with only 10^4 cycles. With this modification, the curves can be observed to be saturating below that new



(a) Result of the Markov Chain analysis.

(b) Result of the simulation.

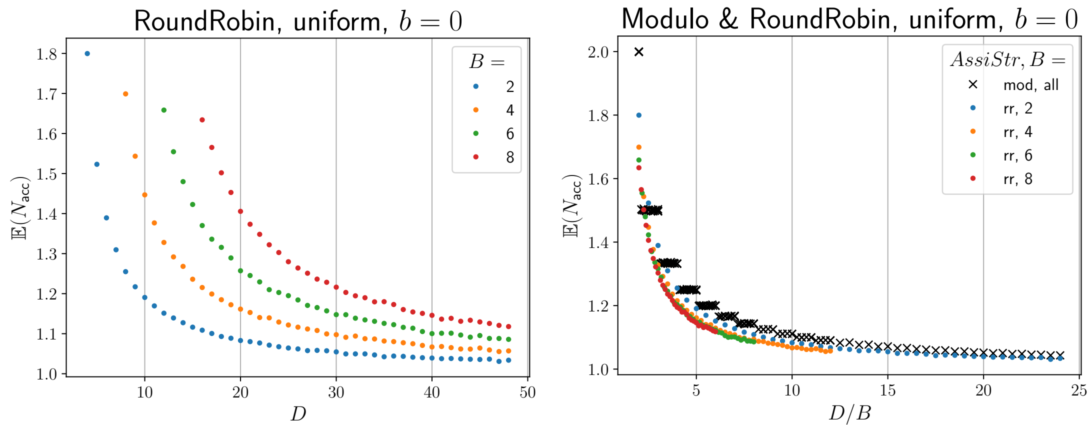
Figure 6.13: Comparison of results from the mathematical analysis and the simulation with same parameter-constraints. The plots show the accumulation of events with normal-distributed destinations under Modulo assignment.

number of simulation cycles.

6.4.2 Simulating Round Robin Assignment

After having verified the method by simulating the Modulo assignment and comparing it to the previous numerical results, now the behaviour of RoundRobin assignment (AS.3 on page 79) shall be simulated. The respective simulation results are presented in Figure 6.14. It shows the simulation of uniformly distributed destinations using RoundRobin assignment. While Figure 6.14b plots the expected accumulation length $\mathbb{E}(N_{acc})$ against the absolute number of destinations D, Figure 6.14a shows the same data, plotted against the relative number of destinations per bucket D/B. The direct comparison to previously simulated Modulo assignment in Figure 6.14b shows very similar behaviour for both strategies. Effectively, RoundRobin assignment performs as a lower bound to the Modulo assignment and without the stepped course. This is expected, as the modulo strategy only assigns another destination to a particular bucket, if in steps of $\Delta D = B$ new destinations, i.e. if $(D-b)\%B = 0$. Therefore with Modulo assignment, $\Delta D < B$ additional destinations perform equally good as the *original* number of destinations at $\Delta D = 0$. However, this effect does not occur with RoundRobin assignment, leading to a continuous degradation in accumulation length performance.

Repeating the simulation with RoundRobin assignment and normally distributed event destinations, leads to the results shown in Figure 6.15. It can be seen that in contrast to the uniformly distributed destinations, now also an $\mathbb{E}(N_{acc}) > 2$ is possible. Actually, the expected accumulation length can now reach very high numbers of more than 10^3, depending on the relative distribution width compared to the number of buckets in the system. The reason for this is that conflicts now become significantly less likely, once all the frequently occurring destinations are assigned to a bucket and the remaining possible destinations are outside the $3\,\sigma$ interval of the distribution. However, this is



(a) Plot of the expected accumulation length $\mathbb{E}(N_{\text{acc}})$ over the *absolute* number of destinations D for different numbers of buckets B.

(b) Plot of the expected accumulation length $\mathbb{E}(N_{\text{acc}})$ over the *relative* number of destinations per number of buckets D/B for different numbers of buckets B. The results from the Modulo assignment are included again for comparison.

Figure 6.14: Results of simulating the RoundRobin assignment with uniformly distributed destinations.

still significantly worse than the previous result of the numerical analysis of modulo-assigning normally distributed destinations in Section 6.3.3 and Figure 6.7, where $\mathbb{E}(N_{\text{acc}}) > 10^{12}$ can be reached around $B/\sigma = 8$. This is, as already hypothesised in Section 6.1.2.3, expected. With the dynamic strategy of RoundRobin assignment, every bucket will eventually become disturbed by a frequent destination, in contrast to the static strategy of Modulo assignment, where only some buckets will frequently become disturbed. This leads to very high performance for some buckets and quite lousy performance (comparable to RoundRobin assignment) for the others. However, it should be noted that any expected accumulation length above the interconnection network's MTU is only of theoretical relevance, as in that case C.1 on page 77 will apply and limit the actual accumulation length. As will be derived in Section 7.3.3 on page 121 a full packet will be able to transport between 62 and 124 events.

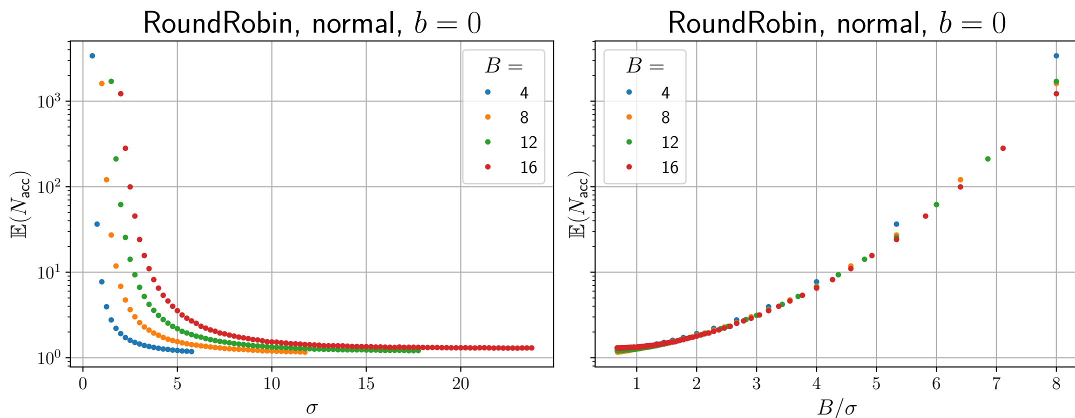


Figure 6.15: Logarithmic plot of the expected accumulation length $\mathbb{E}(N_{\text{acc}})$ with normally distributed destinations over the absolute distribution width (**left**), and the relative quantity of buckets per distribution width B/σ (**right**).

Part III

Implementation and Experiments

7 The Implemented Event Communication

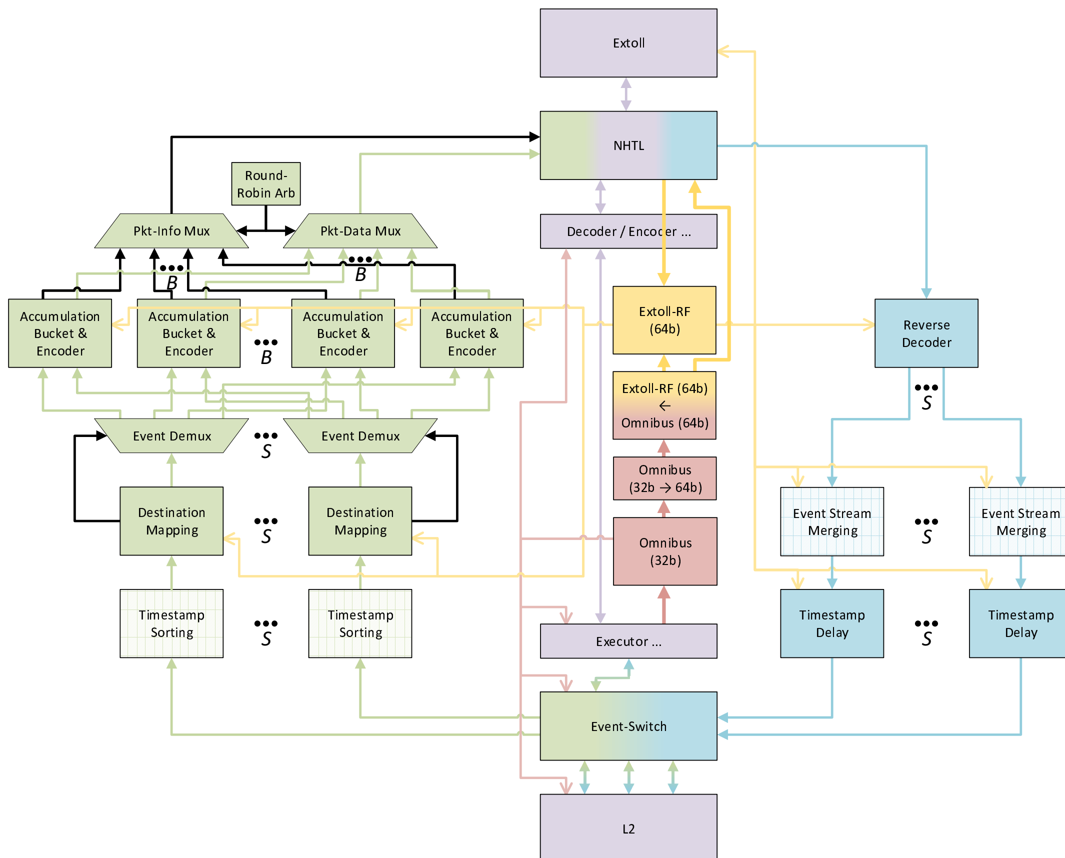


Figure 7.1: Schematic overview of the implemented event communication architecture. The parts responsible for transmission (towards the network) of events is coloured **green** while the receiving parts (events from the network) are coloured **blue**. The shaded modules (Sorting and Merging) are currently not implemented. The two configuration buses are coloured **red** (Omnibus) and **yellow** (Extoll-RF). Modules, previously existing in the original FPGA design are coloured **purple**.

The event communication architecture, implemented and tested in the course of this thesis will be described in detail in this Chapter. Figure 7.1 shows a schematic block diagram of the implemented design. This includes the functionality, previously referred to in Section 3.2 for the **green** boxes in Figure 3.4. The description of the design will start at the bottom of the Figure with the *Event Switch*, as this is the central point in the FPGA where the events are distributed between the different sources and sinks in the design. Second, the concept of how the systime is synchronised across multiple network nodes will be addressed, as this is essential to the global interpretation of spike event timestamps. The description will then follow the course of the events towards and across the network, and on the receiving side back to the target FPGA's *Event Switch*. As the BSS-2 chip

can deliver up to two events per clock cycle on average as stated in (Karasenko 2020), the event communication architecture must be able to process at least two events in parallel. This constraint is honoured by the implementation of multiple parallel data paths, as indicated in Figure 7.1 by multiple parallel instantiations of the respective units. Last but not least, the details about the status and configuration interfaces, namely Omnibus and the EXTOLL registerfile will be described, also including the bridging interface between these two configuration paths. Details on the parametrised implementation of the described design with respect to the number of *Accumulation Buckets* and the number of parallel data paths (*Splits*) will be presented in Section 7.8.

7.1 The Event Switch

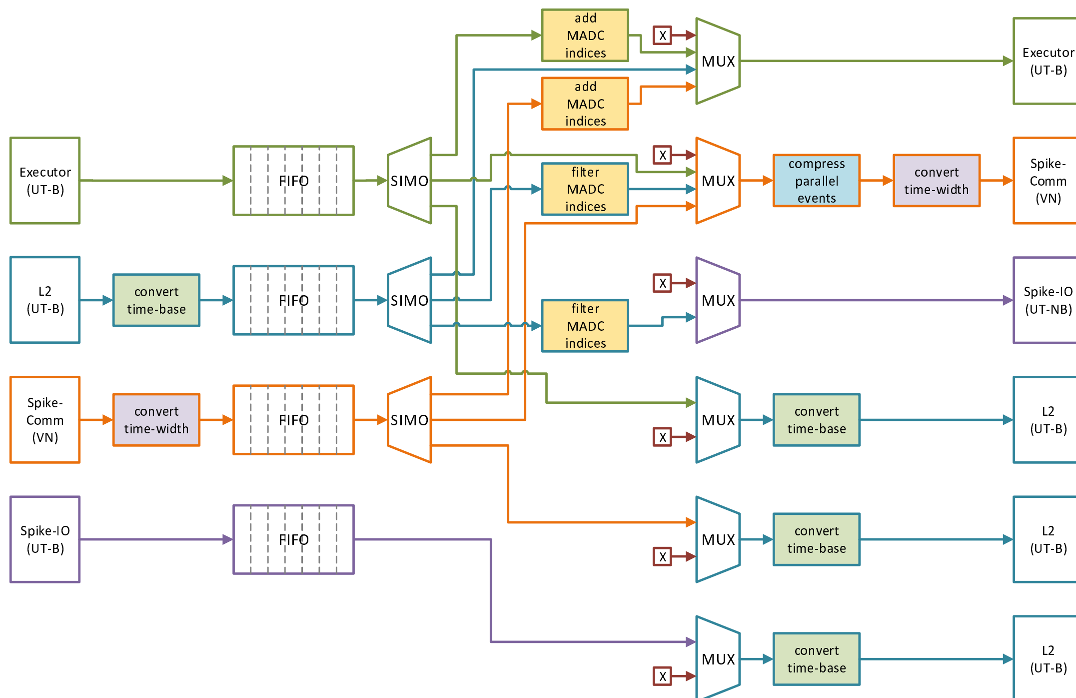


Figure 7.2: Schematic block diagram of the *Event Switch*. The different data paths are colour-coded with respect to their source or sink. Additionally, the different conversion units are coloured with according to the respective conversion operation, they perform.

The *Event Switch* takes event streams from four input interfaces and distributes them to one or more of six output interfaces, a schematic block diagram is shown in Figure 7.2.
The input interfaces collect events from the *Playback Executor* (cf. Section 3.2.2), the *L2* (cf. Section 3.2.1) and the external *Event Communication* (described in this Chapter), as well as the physical spike-io interface, which is used in the work of Yannik Stradmann for directly providing spike output to external (robotic) devices (Stradmann et al. 2023). For each output, the user can statically select which input it shall listen on. An exception of this configuration space are the outputs towards the *L2*, as this interface is replicated three times, to simultaneously take events from the other three inputs. Merging of events towards the *L2*, i.e. towards the HICANN-X chip is implemented through the merger-matrix introduced by (Kanzleiter 2018). The user can also decide to disconnect the outputs independently of each other. However, the three parallel outputs towards the *L2* can only be

switched off together.
In between these interfaces, several conversion units are placed in the data paths in order to shape the event stream for the respective units at the other end. FIFO buffers are inserted into the data paths in order to break the critical timing paths at the flow control backpressure signal (`next`) that otherwise propagates as combinatorial logic through all the pipeline stages of the involved units. A FIFO buffer effectively separates this signal path by providing `full` and `empty` signals that are derived logically independent from each other.
The following Subsections will discuss the details on the respective conversion units.
The *Event Switch* design uses different signal-level interfaces. These include two kinds of *blocking* (`UT-B` in Figure 7.2) and *non-blocking* (`UT-NB` in Figure 7.2) UT interfaces, as well as *valid-next* (`VN` in Figure 7.2) interfaces and *FIFO* interfaces (at the `FIFO` blocks in Figure 7.2). These interfaces and conversions between them will be presented in Appendix B.1.

7.1.1 Index Filtering and Manipulation

The event stream from the HICANN-X chip not only contains spike events, but also analogue sample data from the MADC unit (cf. Section 3.1.1 on page 35 for reference). These are also transmitted in tuples of up to three parallel samples and are distinguished from spike events through their unique UT indices (for details on the UT cf. Section 7.3.3). As these MADC samples are only of interest for *trace* collection in the *Executor* unit, they have to be filtered out of the event streams heading for *Spike-IO* and external *Event Communication*.
The indices at the *L2* interface (i_{L2}) are arranged in a way such that the MADC samples are ranged below the spike events. So filtering for spike events only (i_{spike}), involves checking whether the index lies above the offset-value (o_{MADC}) of the MADC sample types and finally subtracting this offset:

$$i_{\mathrm{spike}} = \begin{cases} i_{\mathrm{L2}} - o_{\mathrm{MADC}} & \text{if } i_{\mathrm{L2}} \geq o_{\mathrm{MADC}} \\ \text{drop} & \text{else} \end{cases} \tag{7.1}$$

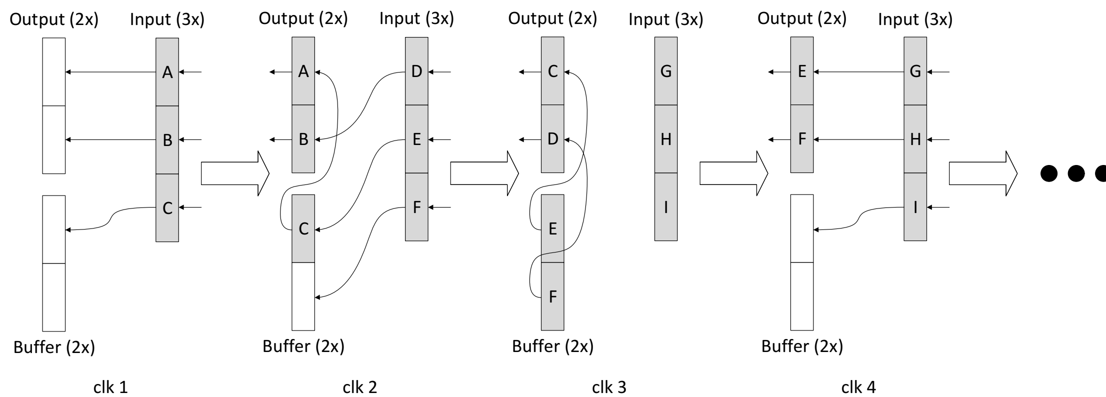
If the incoming index indicates MADC samples, the data should simply be dropped by holding the `valid` signal low. Because of this index arrangement, the indices have to be elevated by the said offset o_{MADC} for events travelling towards the *Executor* unit and not originating from the *L2* interface.

$$i_{\mathrm{exec}} = i_{\mathrm{spike}} + o_{MADC} \tag{7.2}$$

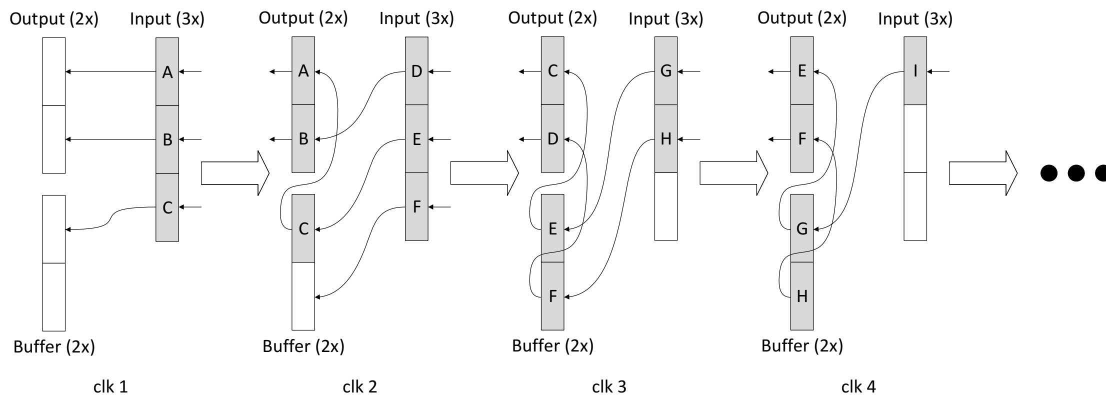
Although these events streams do not contain MADC samples, this conversion is still necessary, in order that the *Executor* unit can correctly interpret them as spike events. Otherwise it would falsely interpret them all as MADC samples. The units performing this filtering and fixing operation are highlighted with **yellow background** in Figure 7.2.

7.1.2 Event Path Compression

In the BrainScaleS-2 event communication system there are different numbers of parallel event-interface ports that are attached to the *Event Switch* unit. The *internal* event interfaces between the *L2*, the *Playback Executor* and the *Spike-IO* are three-way parallel, i.e. they can transport up to



(a) Exemplary data flow with stall condition.



(b) Exemplary data flow without stall condition.

Figure 7.3: Data flow diagrams of the Event-Compressor unit. The Figure shows the states of the data pipeline registers at subsequent clock cycles. The movement of data units through the registers is shown with arrows between the current location and where the data will be after the next clock edge.

three events per clock cycle in parallel. The actual number of valid parallel events is thereby coded into the `idx` field of the respective UT interface using a binary number code. The valid events are always aligned to the least significant position in the overall data bus, so an index value of `idx = 0` means that only the zeroth position of three possible events is valid while an index value of `idx = 1` means that the first two events are valid.

On average there will be two valid events per clock cycle, transported across the serial links (cf. Section 3.1.2 on page 37 for reference). In order to save FPGA resources and keep the design simple, it was decided to only provide two parallel data paths for the external event communication. Hence, the three *internal* data paths have to be compressed onto two *external* data paths. The unit performing this compression operation is highlighted with **blue background** in Figure 7.2. Decompressing the received event stream on the other side of the network transmission is however not necessary, as it would not gain a benefit in the bandwidth usage on the serial links. It is also not wise, as it would add additional design complexity, using up more precious FPGA recourses. Finally, it is not even accurately possible, due to the lack of information about which double events have been compressed triple events and which not.

A data flow diagram of the respective compressor unit as it is implemented in the design is shown

in Figure 7.3. At each clock cycle it takes up to three alignment units (events) of data at the input and outputs up to two previously buffered alignment units. Input data that is larger than the available output space is partially buffered and split for output during two subsequent clock cycles. The output- and buffer registers thereby act as a single barrel shifter that can move up to two data units from the buffer- to the output registers and takes input in the size of up to three units to that exact point of the overall buffer, where it fits without overwriting stored data units. In this configuration, the unit can compress two subsequent triple input data words into three subsequent double output data words. A stall condition will occur at the third triple input data word in a row which can not be buffered anymore and has to wait until the buffer has been (partially) cleared through the output. However, double and single input words can always be taken by the unit, as long as the output is not stalled by the following unit. While double inputs will only postpone an imminent stall condition to the occurrence of the next triple input, a single input will relax the situation such that the unit will be able to take a triple input in the next cycle again.
By increasing the amount of buffer space, the number of triple inputs that may be taken until a stall condition occurs can be increased. The additional buffer space however, is not required to be added in the form of additional barrel shifting registers. Instead it is sufficient and probably more hardware-efficient to add a FIFO memory at the input. One should however notice that increasing the buffer space will only increase the burst capacity until the first stall occurs. From that point on the situation is the same as with minimal buffer space, as long as the situation is not relaxed by low input traffic. Also, the increased buffer space will not improve the overall latency for burst data as the bottleneck is still posed by the output width. However, a larger buffer can decrease the probability of stalling preceding units at a given average burst size.

7.1.3 Timestamp Extension

In contrast to the serial links connecting the chip with its communication FPGA, latencies across the EXTOLL network and through the event communication architecture are expected to reach significantly larger time-spans. Especially these latencies across the network will depend on the distance between the communicating nodes. In case of multicast communication the highest latency between the source- and the most distant target node will come to effect.

```
function st_t ts_to_st (st_t systime, ts_t timestamp);
  st_t timestamp_st;
  timestamp_st = systime;
  timestamp_st[$high(ts_t)-1:0] = timestamp;
  if (timestamp_st > systime) begin
    timestamp_st -= (2**$high(ts_t));
  end
  return timestamp_st;
endfunction
```

Listing 7.1: Conversion procedure for a shortened timestamp back to the original systime-width, based on an evolved systime.

The question of interest here is: How long may a timestamp of a certain size lie in the past, for still being able to uniquely convert it to the larger systime width?

In order to answer this question, let us take a look at the procedure of converting a shortened timestamp back to the wider systime in Listing 7.1. First, the current `systime` is simply assigned to the timestamp conversion result where the lower bits are replaced with the actual timestamp. After this simple replacement, it is checked whether the resulting value is larger than the current `systime`. By assuming that the timestamp must have been created in the past, it is inferred that the converted value cannot lie in the future. Consequently, the reason for the converted timestamp being in the future is assumed to be that the systime counter has bit-overflown the width of the timestamp. Thus, this can be corrected by subtracting the value of this overflow, being `2**$high(ts_t)`. Now this is the answer to the question in the first place, as this correction only leads to a correct value if the overflow has only occured once since the timestamp was created. Moreover, it cannot be known whether this assumption is true, so it has to be assured by selecting the width of the timestamp large enough that during its expected transmission latency the overflow can in any case only happen once. In other words, the timestamp should be at least doulbe as wide as the worst expected latency.

For the BSS-2 chip to FPGA links, the timestamp was designed to be 8 bit wide, corresponding to a maximum allowed latency of around $\frac{2^8 \cdot 8\,\text{ns}}{2} \approx 1\,\mu\text{s}$. For the external event communication in the scope of this thesis, it was decided to transmit timestamps at a width of 15 bit, thus being on the safe side for latencies of up to $\frac{2^{15} \cdot 8\,\text{ns}}{2} \approx 130\,\mu\text{s}$, corresponding to a biological timescale of 130 ms (cf. Section 5.3.1 on page 62). These axonal delays of course include the accumulation and transmission latencies (cf. Equation (5.4)).

The resulting conversion of 8 bit timestamps from the BSS-2 chip to 15 bit timestamps for the external event communication can basically be done by the procedure in Listing 7.1. As the systime is 43 bit wide, the incoming timestamps can be upscaled to that width and then only select the lower 15 bit. On the other side, the timestamps have to be re-shortened back to 8 bit for the target chip, which however is only another bit-select operation.

The timestamp width conversion units are depicted with **purple background** in Figure 7.2. Together with the 14 bit address label, the overall event format is now 29 bit wide and is summarised in Listing 7.2.

```
typedef logic [13:0] event_t;
typedef logic [14:0] timestamp_t;
typedef struct packed {
  event_t     event_address;
  timestamp_t timestamp;
} timed_event_t;
```

Listing 7.2: The spike event format, as used for network transmission.

7.1.4 Global Time-Base Conversion

When communicating spike events with target timestamps through a system with multiple asynchronous systime clocks, these timestamps are barely interpretable at the destination systime, given their creation in the source systime.

A time-base conversions takes place at the interfaces from and to the *L2* in order to convert the timestamps of incoming and outgoing between the local and a global systime base. This makes event

timestamps globally interpretable in the whole interconnected system. The respective conversion units are highlighted with **green background** in Figure 7.2.

The implemented method of synchronising the global systime will be presented in detail in Section 7.2. Basically, an offset value is subtracted from the incoming events from the chip and re-added to the events going down to the chip (cf. Figure 7.4). This offset value will be different at each system unit (chip plus FPGA). In order to synchronise the whole system, it is important that these offset values are determined at the same point in time at all FPGAs. Because the offset value is stored in systime units at a width of 43 bit, the timestamps are first scaled up as in Section 7.1.3, then the offset is applied and finally the timestamps are re-shortened to the original width of 8 bit. However, this width-conversion is expected to be optimised away by the FPGA design compiler, as it does not have a persistent effect due to the subsequent bit-select operation.

7.1.5 Event Replication

```
always_ff (@posedge clk) begin
  reg_out <= reg_in;
end

always_comb begin
  reg_in = reg_out;
  in.next = 1'b0;
  for (int i=0; i<num_ifs; i++) begin
    if (out[i].next)
      reg_in.valid_out[i] = 1'b0;
  end
  if ((reg_in.valid_out == '0) && in.valid) begin
    reg_in.valid_out = '1;
    for (int i=0; i<num_ifs; i++) begin
      reg_in.data_out[i] = in.data;
    end
    in.next = 1'b1;
  end
end
```

Listing 7.3: Pseudo-Code for replicating one Valid-Next data stream to `num_ifs` independent Valid-Next data streams. The input is blocked, until all outputs have accepted the data.

As described at Section 7.1 on page 110, the working principle of the *Event Switch* is that the user can select an input-port for each output-port from which to forward events to that output. Consequently, it is possible to select the same input for more than one output. This implies the necessity to replicate input event streams to multiple output ports. This replication is depicted in Figure 7.2 at the *Single-In-Multiple-Out (SIMO)*-Blocks. The difficulty here is that the output interfaces are independent of each other and can thus acknowledge the acceptance of valid input data at different points in time. The `valid` signal at the respective outports has to be de-asserted immediately after they have individually acknowledged the data. Otherwise, the output ports would read the same data multiple times. However, at the input interface, a valid event datagram may only be acknowledge, when all output interfaces have acknowledged the datagram. Otherwise, some output ports could miss a valid datagram. A pseudo code implementation of these constraints is shown in Listing 7.3. Unfortunately however, this leads to the behaviour that a single blocked output will block all other outputs too, as

the input is blocked. This might become a problem through the prioritised sending of live-events with respect to trace-data at the NHTL (cf. Section 7.4). Mitigation strategies for this problem will be discussed in Section 8.2.2 on page 155.

7.2 Systime Synchronisation

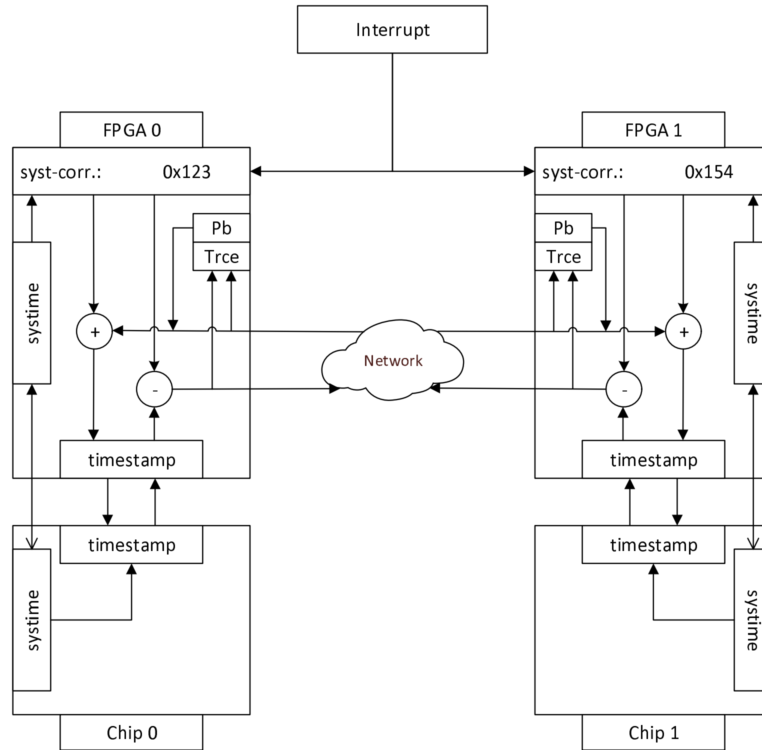


Figure 7.4: Schematic block diagram of the control flow for the system-time synchronisation.

As mentioned above in Section 7.1.4, a way to synchronise multiple BSS-2 systems with respect to their systime is needed in order to create a larger system, exchanging spikes between multiple accelerated neuromorphic chips. The following Subsections will go into the details, of how the systime is synchronised, first between each chip and its attached FPGA, and second between all the FPGAs performing a multi-chip experiment. The overall synchronisation procedure is visualised in Figure 7.4.

7.2.1 Local Systime Synchronisation

The mechanism for synchronising the systime between the BrainScaleS-2 FPGA and the attached ASIC was originally described in (Rettig 2019a). The FPGA thereby requests the chip to reset its systime to a configurable value and send back an acknowledgment, containing its current systime value. The FPGA measures the round-trip latency (l_{rtt}) of request and response and then resets its own systime to the appropriate value (which is visualised by an open arrow-head towards the chip's systime block in Figure 7.4 and a filled arrow-head in the opposite direction). As the BSS-2 chip runs at double the clock speed of the FPGA, also the ASIC systime counter advances with double

speed as compared to its FPGA counterpart. This, and the fact that the serial links have a directional latency-offset of around 5 FPGA clock cycles (determined by simulation in (Rettig 2019a)), leads to the following equation to determine the correct FPGA systime:

$$t_{\text{fpga}} = \frac{l_{\text{rtt}}}{2} + 5 + \frac{t_{\text{asic}}}{2} \tag{7.3}$$

Whether the systime is reset or just reported back is determined through a single payload bit in the request message to the chip and the respective playback command that triggers this synchronisation process. The reported ASIC systime, as well as the measured round-trip latency are in any case stored at status registers, accessible through the FPGA Omnibus.

7.2.2 Global Systime Synchronisation

With the mechanism described above, the FPGAs and ASICs are synchronised pairwise to each other. The objective now is, to find a way to synchronise these islands of local systimes to achieve a globally synchronised systime.

The EXTOLL network provides global barrier- and interrupt operations (Burkhardt 2007, 2012) which are used for this purpose in this work. The operation principle of the respective hardware support units is described in Section 4.1.2. Originally, the status of the interrupt- and especially the barrier operation, i.e. whether it has been released or not, has to be polled from software at the respective status-registerfile. The interrupt notification will additionally trigger an automatic kernel interrupt in the operating system of a participating host computer. However, as the interrupt- and barrier operations are to be used in an FPGA implementation, an FSM-based hardware unit doing the control and status polling operation on the barrier unit has been implemented. The control and status interfaces have been attached to the *Playback executor* and two additional *wait-until* instructions to the playback instruction-set have been added (cf. Section 3.2.2). The **Wait-until-barrier** instruction will notify the *Barrier Unit* that the playback program has *reached* the barrier point and block the program until the unit reports to be in *released* state. The **Wait-unit-interrupt** instruction will simply block the execution until the *Interrupt Unit* reports an interrupt notification. This interrupt notification signal is thereby hard-wired to the *Playback Executor* unit. This will signal a common point in time between all participating units in the configured interrupt tree. This interrupt operation however relies on a single FPGA executing a master playback program that has to start the interrupt operation through a write-access to the control registerfile of the root-node's Interrupt Unit.

The actual quality of the *Global Interrupt* mechanism in terms of time-jitter is experimentally determined later in Section 8.6 of this thesis.

This *Global Interrupt* can now be used to create a globally synchronised systime across all FPGAs. For this, the hardware interrupt signal from the *Interrupt* unit is connected to the *Event Switch* unit. Here the unit listenes on that signal and uses it to synchronously reset the global systime counter. Simultaneously the offset to the locally synchronised chip-systime is synchronously stored in a register. This value is used for converting the incoming and outgoing event timestamps from and to the BSS-2 ASIC by adding or subtracting this offset, depending on the event's direction (cf. Section 7.1.4).

7.3 Event Transmission

If the *Event switch* forwards events towards the external communication network, they enter the SPIKE_COMM partition in the FPGA design (cf. Figure 3.4). This partition contains all the design units that are necessary to pack and un-pack spike events to and from network packets. This Section will go into the implementation details of the sending-, i.e. packing side.

7.3.1 Timestamp Sorting

In order to be able to delay events at the destination FPGA until their timestamp is due without over-delaying subsequent events that have a smaller timestamp, events have to be transmitted in sorted order. As events are timestamped sequentially at the L1-to-L2 interfaces in the HICANN-X chip, the outgoing event stream at the BSS-2 ASIC is already sorted. However, because this event stream is transported in parallel through 8 serial communication links between the chip and the FPGA events may be disarranged, i.e. jittered, to an amount of around 8 to 10 clock cycles due to implementation details of the mechanism distributing the events across the parallel links. With a clock period of 8 ns, this corresponds to around 64 ns to 80 ns of jitter in the hardware time-domain or µs in the biological time-domain respectively. When comparing this to the STDP time constants (cf. Section 5.3.2 and (Friedmann et al. 2017)), in the worst case (fastest time constant and highest jitter), this corresponds to 0.7 % of that time constant and is thereby considered in an acceptable range. Consequently in this first implementation, the timestamps coming from the chip are not sorted (the module is highlighted with a shaded grid in Figure 7.1). When the (also not yet implemented) merging of event streams (Section 7.5.2) is added at the receiving branch in a future improvement, the sorting operation could be implemented completely at the merging units (cf. Section 9.2 on page 194).

7.3.2 Destination Mapping

The first important task, implemented in the event communication architecture at hand, is the mapping of the source neuron labels to destination synapse ids and assigning network target node ids to the events. Events arriving from the L2 at the FPGA and forwarded through *Event Switch* have a bit-structure, as shown in Listing 7.4:

```
typedef logic [ 1:0] l1_adr_t;
typedef logic [13:0] nrn_adr_t;
typedef struct packed {
  l1_adr_t  l1_address;
  nrn_adr_t neuron_address;
} l2_event_t;
```

Listing 7.4: Address label format of L2 events from and to the BSS-2 HICANN-X ASIC.

The two most significant bits, forming the `l1_address`, indicate at which of the four L1-to-L2 event interfaces the respective event has been processed. However, this information is not quite useful for mapping to a destination synapse address, as all neurons on the chip can be configured to be routed across any of those event interfaces and incoming events at the chip can also reach all synapses across any of the event interfaces. Therefore, these two bits are ignored for the mapping

operation and not transmitted across the network. The receiving FPGA will however have to re-add them, as to comply to the correct format of events towards the receiving HICANN-X chip. The actual decision of which L1 event interface is to take the events at the receiving chip can be made by the target FPGA, based on the rate of events from the network.
The mapping operation is implemented using a large lookup table, addressed directly by the remaining 14 bit `neuron_address` of the incoming L2 events. This 14 bit address space thereby leads to a lookup table with 16,384 entries. Each of these entries contains data of the shape, presented in Listing 7.5 and a total width of 18 bit.

```
typedef logic [13:0] nrn_adr_t;
typedef logic [2:0] bkt_id_t;
typedef struct packed {
  logic     valid;
  nrn_adr_t dst_event;
  bkt_id_t  bucket_id;
} lookup_entry_t;
```

Listing 7.5: Format of lookup table entries in the implemented event communication architecture on the BSS-2 FPGA.

The individual entries contain a `valid` flag, the 14 bit target synpase address (`dst_event`) and the index of the bucket unit which is to accumulate events from the respective source. The width of this `bucket_id` is determined by the number of bucket units implemented in the design. For a number of 8 buckets, this field is 3 bit wide. This leads to an overall memory requirement of 288 kbit (1 kbit = 1024 bit).
Care is taken that the implementation code will result in the usage of block-RAM resources in the FPGA, as this is the most efficient way to implement large amounts of memory. In the Xilinx®7 series FPGAs, which is used in the BrainScaleS neuromorphic hardware platform, each 36 kbit block RAM offers 2 kE at a width of 18 bit (Xilinx Inc. 2019). The naive expectation would therefore be that the physical implementation of the lookup table will use 8 of these 36 kbit block RAMs. However, it turns out that rather than combining the address space of several wider block RAMs, the synthesis tool prefers to combine several narrow block RAMs, each supporting the full address space. This considerably reduces the logic resources required to multiplex the address-space across the block RAMs. Therefore, the physical synthesis tool actually uses 9 block RAMs at an individual width of 2 bit to reach the required entry width of 18 bit (cf. Section 8.2.1). Each block RAM resource on the FPGA offers 16 kE of this size, directly matching the required address space.
The lookup table memory is included into the configuration registerfile. Thereby, it can be configured through the common configuration interface from each node in the network.
As the event communication architecture handles multiple events in parallel, the destination mapping unit also has to be instantiated this number of times. It should be noted that due to limitations of the registerfile generator tool (cf. Section 4.2.4) each of these instantiations has to be individually configured at its own address-space in the registerfile. Details on the design considerations regarding these parametrisations will be given in Section 7.8. The experiment software on its turn will have to configure the mapping entries multiple times onto the hardware. This is represented in the hierarchical structure of the coordinate classes, as described in Section 8.4.4. As this significantly

impacts the required time to configure the system, this should be a subject to future optimisation (cf. Section 9.2 on page 195). However, the configuration is not considered critical with regard to its execution time, as it does not impact the realtime experiment performance.

7.3.3 Accumulation Buckets

After the spike events have now been mapped from source neuron ids to target synapse addresses, they are *demultiplexed* to the respective accumulation bucket, as was indicated by the respective lookup entry.

The main purpose of these buckets, as defined in Section 6.1, is to accumulate spike events and form larger packets in order to reduce the header overhead before transmitting them across the network.

In the current implementation at hand, the *Accumulation Buckets* are statically configured for a specific 16 bit network destination node-address, or multicast group-id (cf. Section 4.1.3 on page 51). A boolean configuration flag defines, whether the destination address is to be interpreted as node-id or muticast group-id. An axonal transmission delay of up to 2^{14} clock cycles ($\approx 130\,\mu s$, cf. Section 7.1.3) is also statically configured per bucket and added to the events' timestamps.

Events are accumulated until the aggregated packet is full, or one of two statically configured time-outs applies. The first timeout counter measures the time between two subsequent events entering the bucket. In contrast, the second timeout counter measures the total time, the first event of a packet has been stored in the bucket. Under the assumption that the events' timestamps arrive in ascending order, or at least have a small jitter, this timeout condition is a good approximation to the Condition C.2 on page 77. Both of these timeout values are configurable to an extent of up to 1024 clock cycles, corresponding to $\approx 8\,\mu s$ hardware time and $\approx 8\,ms$ bio time.

By configuring the network destination addresses statically in the buckets instead of individually for each source neuron address in the lookup table of Section 7.3.2, the number of possible destinations is effectively restricted to exactly the number of available buckets. Thereby, destination conflicts as analysed in Chapter 6 are principally excluded from occurring by design. This ensures an optimal aggregation until either the packet is full, or the timeout exceeds. Of course this is only possible as long as the restricted number of possible destinations suffices the modelled spiking neural network or the FPGA resources suffice for increasing the number of implemented buckets.

As the design has to process at least two event streams in parallel, as explained in Section 7.1.2, it will generally happen that more than one mapping unit (Section 7.3.2) request the same bucket. The bucket unit therefore either has to arbitrate the right of access between the requesting mapping units, or be able to accept all the incoming events in parallel. As an arbitrated access to the accumulation buckets would inevitably decrease the throughput of the overall design, the second solution was selected for the implementation at hand.

Event data is encoded into the network packet using the UT *Encoder* which was introduced by (Karasenko 2020). As input, it takes a blocking UT interface (cf. Listing B.2a). The data-index pairs at the input are then flexibly encoded and serialised into output datagrams of a fixed width. For this task the UT *Encoder* uses parametrised information about the widths of the individual data types, transported across the input interface and distinguished by their respective index value. In (Karasenko 2020), this information defines the UT *alphabet*. If the widest element on the alphabet is wider than the output width, the input data is serialised onto multiple output datagrams. However,

if the widest element on the alphabet is shorter than the output width, the data is not de-serialised, but rather padded to the full output width. This fact makes those parametrisations inefficient, as it leads to possibly high portions of wasted bandwidth on the physical link. On the other hand, parametrising the UT *Encoder* output narrower than the width of the physical data output for a serialising operation is also inefficient. Thereby, multiple shorter datagrams are concatenated to the physical data bus, effectively reducing the throughput of the UT *Encoder*, as it outputs one datagram per clock cycle. So all in all, it is desirable to have the *Encoder* output as narrow as possible in order to not waste too much encoding space, but also as wide as necessary to fit a whole input index-data pair. If the physical bus width is very large compared to the optimal *Encoder* output width, it might even be possible to stack multiple UT datagrams into a single physical output datagram. Thereby, only the transmission latency of the datagrams is partially increased, but the throughput is maximised and the available space in the network packet is optimally used.
In the case of this design, the UT alphabet has to encode the number of valid events, processed in parallel at the current clock cycle, as well as their position, i.e. to which event stream they belong. This is achieved by encoding the positions of the currently valid events directly into the binary index representation. As the case of no valid event does not produce an empty datagram, but rather no datagram at all, the positions can be encoded as follows:

$$\texttt{idx} = \texttt{poscode} - 1 \quad . \tag{7.4}$$

So for example with two parallel event streams, this encoding scheme results in a UT *alphabet* as shown in Listing 7.6.

```
localparam logic [2:0][31:0] event_typelist = {
    32'd29,   // 1 valid event at position 'b01
    32'd29,   // 1 valid event at position 'b10
    32'd58    // 2 valid events at positions 'b11
  };
```

Listing 7.6: The event encoding UT *alphabet*, as used in the accumulation buckets for spike event communication across the network. The `event_typelist` parametrises the respective bit-length of UT input-datagrams typed with the respective index.

As the events are 29 bit wide (cf. Listing 7.2), the datagrams are either 29 bit long for one valid event, or 58 bit wide in case of two valid events. The index decrement in Equation (7.4) saves the definition of an alphabet entry with zero width that would never be used by the actual data stream. In this example, the largest datagram is 58 bit wide and therefore safely fits into a 64 bit = 1 QW output width together with an additional 2 bit encoding of the index. As the physical width of the EXTOLL Network-Port on the BrainScaleS FPGA is 128 bit = 2 QW (cf. Thommes 2018), this fits two of these UT output datagrams in a single network datagram.
After the UT encoding stage, the encoded data stream is kept in a FIFO buffer to perform the actual packet aggregation task. As the MTU of the EXTOLL network is 62 QW of payload data (cf. Figure B.1), a full packet will contain between 62 and 124 encoded events. When a packet is ready for sending, the bucket requests access towards the NHTL network interface amongst its peers. When granted by round-robin priority, the buffered data is shifted out towards the NHTL. Header infor-

mation like the final packet size and the network destination address is forwarded across a second interface towards the network together with the last word of packet-data. The access grant is kept by the bucket unit until the last word of the packet has been transferred to the control of the NHTL unit. Then it is released back to the arbiter for another bucket to send its contents in turn.

The buffering scheme in the bucket implementation takes care that the bucket will not be blocked for new input during the request- and flush phase of an outgoing packet. This is done, by implementing two data counters. One counter, which is called the *occupancy counter*, keeps track of the total amount of encoded datagrams, buffered in the bucket. Whenever this counter is higher than the threshold for a full packet, a request is issued for this packet towards the network. During the request and shift-out phases of that packet, the buffer will continue accepting new datagrams until it is full. The second counter, which is called the *packet counter*, keeps track of the amount of datagrams released into the current outgoing packet. This counter ensures that the packet does not exceed the network MTU and determines the packet's actual size, which will defer from the MTU in case of a timeout condition. During shift-out the *occupancy counter* is decremented while the *packet counter* is incremented. If data is accepted simultaneously at the buffer input, the *occupancy counter* will stand still. If it is still higher than the packet threshold after the first packet is completely sent, another packet will be requested.

7.4 The NHTL Transaction Layer

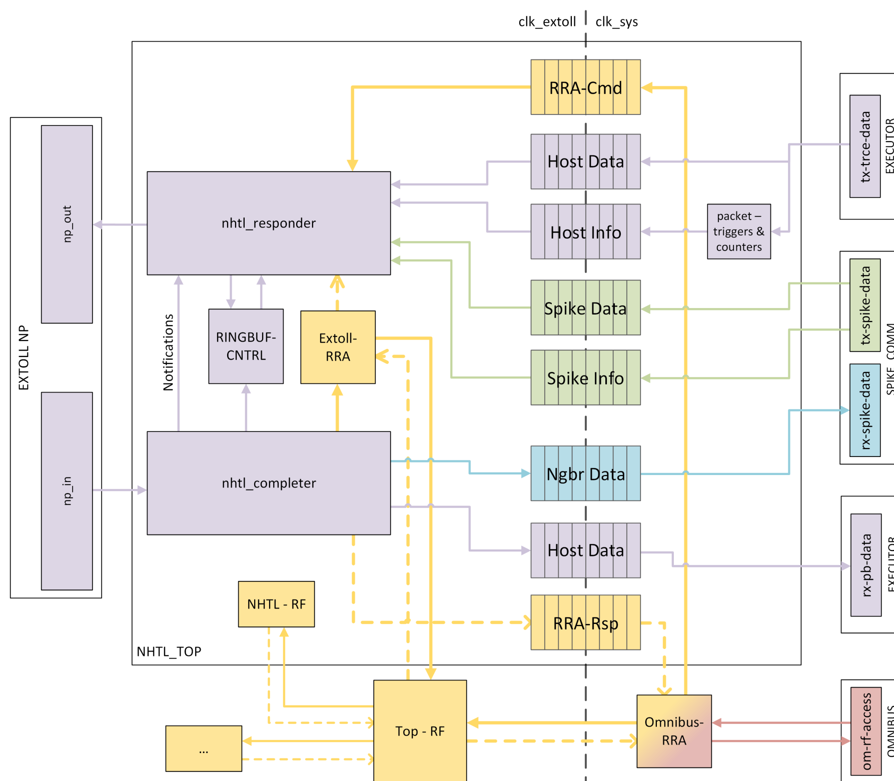


Figure 7.5: Schematic block diagram of the NHTL communication unit. The colour coding is the same as for Figure 7.1

The NHTL unit, which was originally developed in (Thommes 2018) is depicted as a schematic block-diagram in Figure 7.5. Its main purpose is to multiplex the different types of data streams from the BrainScaleS system FPGA onto the EXTOLL Network-Port and demultiplex data coming from the Network-Port towards the respective receiving units. For this purpose it basically implements a simplified version the RMA protocol, as used by the EXTOLL RMA unit (cf. Section 4.2.1 and Appendix B.2).

As already summarised in Section 3.2.4, it receives *playback data* from the experiment control software stack on a host computer, forwarding it to the *Playback Executor* unit. Received *trace data* from the *Playback Executor* is sent to a preconfigured memory region back on the host computer.

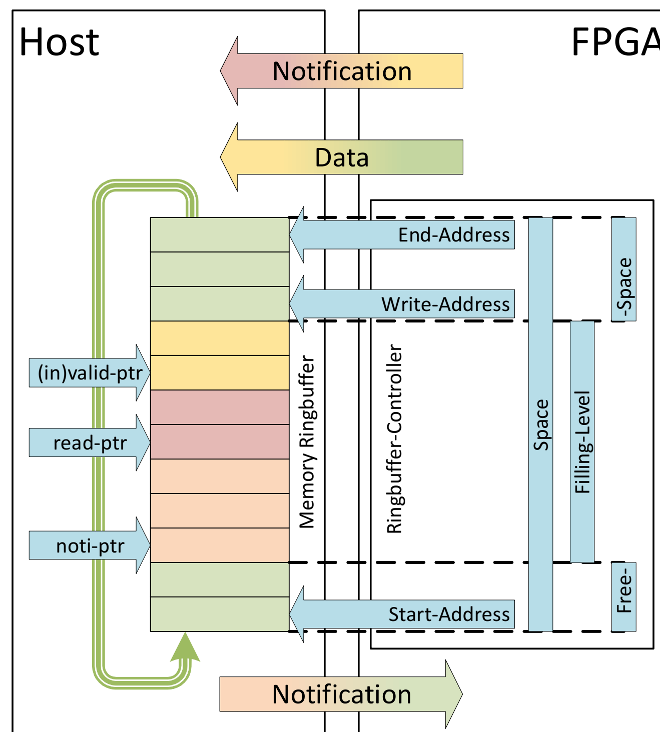


Figure 7.6: Schematic block diagram of the ringbuffer communication scheme between the FPGA's NHTL unit and its software counterpart.

This is done using a ringbuffer communication scheme that basically resembles a simplified VELO protocol for single-ended communication (cf. Section 4.2.2), and does not require to exactly match the existing protocol for compatibility with the existing VELO software library. A block diagram of this communication scheme is depicted in Figure 7.6. The FPGA side keeps track of the current write-pointer and the available space in a reserved memory region on the host. The *trace data* is then directly sent to this memory area using RMA. Dedicated notification messages are used to synchronise the FPGA based write pointer with the host based read pointer. For this, the FPGA informs the host software about the amount of written data and the software acknowledges the FPGA about the amount of data that has successfully been received (for details on the software library, cf. Section 8.4.1). The sub units and data paths, responsible for host-communication in the NHTL unit are coloured **purple** in Figure 7.5. The main difference to the original BSS-1 implementation in (Thommes 2018) is that with BSS-2 the *Application Layer* interface from and to the *Executor* unit (equivalent to the former *Application Layer* in the BSS-1 FPGA design, compare Thommes 2018)

does not provide different data types anymore. Instead it now transports a single UT-encoded data stream that is decoded by the software and vice versa.

In addition to the host-communication, the NHTL now also manages the sending and receiving of spike communication packets that have been pre-assembled by the accumulation buckets (cf. Section 7.3) to and from the network. This is done by sending RMA-put messages directly to the respective destination FPGA, where their payload is forwarded to the receiving side of the `SPIKE_COMM` partition (cf. Section 7.5). The respective data paths are coloured **green** for the sending direction and **blue** for the receiving direction in Figure 7.5, the same as in Figure 7.1. While the sending direction needs to forward some additional information in a separate data path, regarding the desired network destination and the final size of the packet, the receiving side does not require this kind of extra information. Spike communication messages are thereby transmitted with higher priority as compared to host traffic in order to honour the realtime requirements of spike event communication (cf. Section 5.3). On the receiving side, spike event packets are distinguished from playback data packets by their target address field in the packet header, just like the different application data types in the BSS-1 design (cf. Thommes 2018).

The NHTL unit also provides native access to the configuration and status registerfile in a way, compatible to that on the EXTOLL hardware (cf. Section 4.2.4). The sub units and data paths responsible for the registerfile access are coloured **yellow** in Figure 7.5. Master access paths are thereby depicted using straight lines, while slave access responses are visualised using dashed lines and open arrow heads. Remote Registerfile Access (RRA) commands are transported across the network using RMA messages with a special bit-flag set, marking them as targeted towards the registerfile (cf. Figure B.3 and Figure B.4). A remote registerfile can be accessed using write- or read commands, depending on the individual access rights of the respectively addressed register. In addition to the BSS-1 implementation, a second access path was added from the Omnibus. Originating from this Omnibus unit, it is also possible, to access the local registerfile, as well as remote registerfiles of any other node in the EXTOLL network. For this purpose, the NHTL offers a dedicated interface for the injection of RRA commands which are then sent to the respective remote node. The local registerfile itself offers two arbitrated master ports, one for remote access from the network and one for local access from the Omnibus. Through these paths, any Omnibus master (as for example the *Playback Executor*, or the PPUs on the HICANN-X chip) can access any configuration and status register in the network (cf. Section 3.2.5 and Section 7.6). For more details on the implementation of this Omnibus to registerfile bridge, cf. Section 7.6.

7.5 Event Reception

7.5.1 The Reverse Decoder

When spike event packets are received and identified by the NHTL unit on the target FPGA, their payload content is forwarded to the spike communication receiver. Here the incoming stream of network datagrams is disassembled into the parallelly packed UT datagrams. These are then fed through a UT *Decoder* instance, parametrised with the same UT alphabet as the previous encoding stage on the source FPGA (cf. Section 7.3.3). The original parallel event streams from the sending side can now be reconstructed from the decoded UT indices. So if the streams have been in timed

order before (cf. Section 7.3.1), they will remain in that order after reception at the destination FPGA.

When receiving event packets from more than one source FPGA, these packets will always arrive sequentially at the EXTOLL Network-Port. Therefore, it is in principle possible to decode all the incoming event streams from different packets, using a single UT decoder. However, this is practically only possible, if the serial datagrams from individual UT encoders, belonging to the same index-data pair at the encoding input, do not overlap the border between two subsequent packets. Otherwise if might happen that these datagrams were interrupted by another packet originating from a different source and thereby breaking the encoded data stream. As the bucket implementation at hand does not explicitly take care of this, it has to be implicitly ensured that this situation will never arise. This is achieved by carefully parametrising the width of the encoded datagrams large enough, such that a single index-data pair will never be serialised into multiple datagrams. This careful parametrisation of the UT stream width is as well desirable for performance reasons, as explained in detail in 7.3.3.

7.5.2 Merging of Event Streams

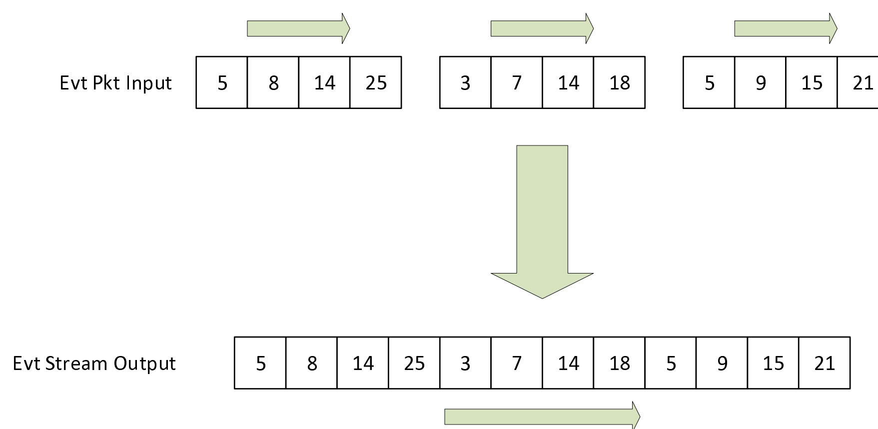


Figure 7.7: Schematic representation of three incoming event packets containing overlapping parts of event streams in sorted order. These have to be merged to form a single overall sorted event stream. **Arrows** depict the direction of time.

When receiving event streams from different sources, these will arrive in subsequent network packets at the target FPGA and decoded as described above. However, despite the global synchronisation of systimes across the network, it is not possible to have the individual event streams arrive in the correct order at a common destination. This is, among other reasons, principally impeded by the heterogeneous timespans across which the events are accumulated into network packets at their source. Another important reason for this are the different transmission delays between the individual source nodes and their common target node. Also, the accumulation processes at different sources will probably overlap with respect to each other. Consequently, although the incoming event streams might individually be sorted, they will almost certainly overlap at their destination.

An exemplary situation for three overlapping input event streams is shown in Figure 7.7. To compensate for this overlap, the incoming event streams have to be merged into the correct order, forming a single sorted stream of spike events that can then be emitted towards the L2 of the attached

HICANN-X chip. This merging will however take some time, as it has to take into account multiple subsequent packets containing event stream snippets. How long this can take, depends on how long the unit waits for additional packets that might contain events which are to be sorted in front of all the events received. The sending side should add a delay margin for this merging operation to the outgoing event timestamps in order to allow for proper merging without inducing large drop counts.

As events are processed in parallel data paths in the communication architecture at hand, each of these data streams will need such a merging unit. The position of these **merging units** is depicted on the right side of Figure 7.1 with shaded background. As the merging problem is a rather complex task, it has not been implemented yet. Therefore, the design at hand only supports the reception of events from a single source node. For a brief overview on the design space of this problem please refer to the Outlook Section 9.2.

7.5.3 The Timestamp Delay Buffer

The received events now have to be delayed until the globally synchronised systime matches their timestamp. Only then they may be forwarded through the *Event Switch* unit towards the HICANN-X. For this purpose, the events are inserted into a FIFO buffer, from where they are only extracted once their timestamps are greater or equal than the continuously evolving value of the systime counter. However, before the events are inserted into the FIFO buffer, they are first checked whether or not they have already timed out during transmission. An event is only inserted into the buffer, if the current systime is still greater or equal than its timestamp. This pre-check optimises the usage of the precious buffer space, as these events would probably be even further delayed, given that the buffer is probably not empty when those events arrive.

These two comparisons between the evolving systime value and the timestamps thereby have to be done carefully with respect to overflow conditions of both operands. This will be elaborated in detail in Section 7.5.3.2.

The behaviour of the *Delay Buffer* unit can be configured for whether to buffer received events or to immediately forward them and if so, whether to dump the timestamp to the current systime or not. These options can be used to configure a mode, where the transmission delay shall be as low as possible and any transmission jitter is tolerated by the model. This is especially useful for latency measurements for the transmission, which can be used to get a first estimate of the required axonal delay (cf. Equation (5.4)). However, as explained in Section 5.3.2, this mode should not be used for STDP related experiments. The HICANN-X chip has a small amount of buffering capability and will add a configurable offset to incoming timestamps and delay them until recent with respect to local systime, in order to settle the jitter across the serial links between FPGA and chip (cf. Section 3.1.2 on page 37). However, as this internal buffer is quite small, receiving events with timestamps far in the future will lead to significant spike loss. Therefore, if forwarded immediately, the timestamps should be dumped to the current systime to prevent this spike event loss on the chip. Besides these configuration options, the unit reports the number of dropped events due to a full buffer memory and expired timestamps respectively.

7.5.3.1 Estimation of Network Latency and Buffer Size

According to Equation (5.10), the required buffer space to avoid event loss due to a full delay buffer is larger or equal the minimum total transmission latency ($\min(l_{\text{tot}})$) including accumulation, network packet transmission and sorting, subtracted from the configured axonal delay. As stated in Section 7.3.3, the implementation at hand allows the axonal delay to be configured in a range up to 2^{14} systime clock cycles, corresponding to around 130 µs. The accumulation time in turn has a lower limit set by the inter-event timeout as configured by the user, according to Section 7.3.3.
For the link latency, (Karasenko 2020) states a range between 268 ns to 512 ns in the direction from FPGA to the chip and between 268 ns to 356 ns from the chip to the FPGA.

$$536\,\text{ns} \leq l_{\text{link}} \leq 868\,\text{ns} \tag{7.5}$$

Finally, the network transmission latency can be estimated by the following considerations: EXTOLL claims a hop latency of around 70 ns on the Tourmalet Network card (cf. Chapter 4 on page 48) which is valid for the original Tourmalet clock frequency of 630 MHz. However the Tourmalet cards used in this project are clocked at a slightly lower frequency of 600 MHz. Additionally, to match the implemented FPGA clock frequency of 100 MHz and only four out of twelve lanes of the link used to connect to the FPGA, the Tourmalet links are configured to run at half clock frequency (300 MHz), whereas only the frequency constraint has an effect on the latency. For details on these clocking considerations cf. Section 8.2.2. Together, these numbers lead to an expected hop latency of around 147 ns on the Tourmalet and 441 ns on the sending and receiving FPGA respectively.
In total, this leads to an expected network transmission latency of

$$l_{\text{trans}} \approx 147\,\text{ns} + 2 \cdot 441\,\text{ns} = 1.029\,\mu\text{s} \tag{7.6}$$

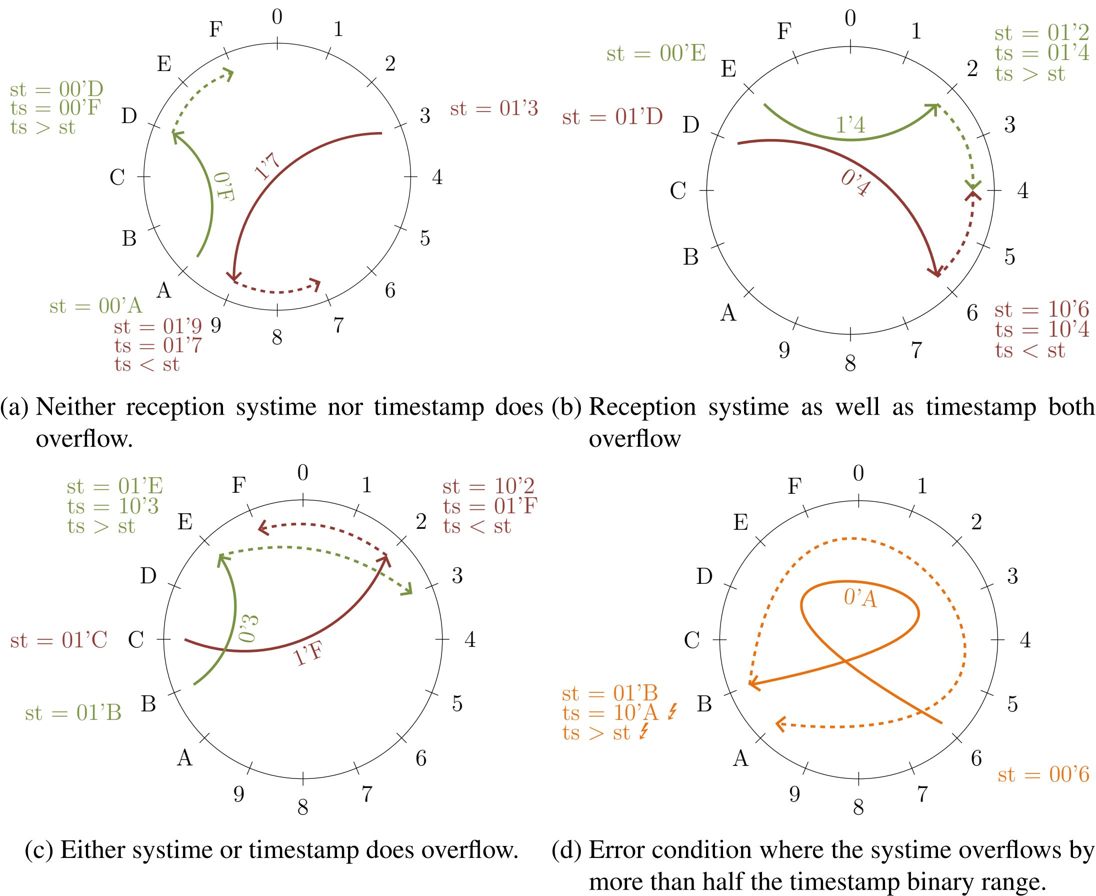
which holds true for a minimal network with one Tourmalet network card. In larger network this on average scales with the network diameter and in particular with the distance between two communicating FPGAs. One should note that for hops between two EXTOLL Tourmalet ASICs, a full 12 lane link can be used at the full speed of 600 MHz leading to additional hop latencies of 73.5 ns. This latency estimation will later be verified by experiment in a minimal network in Section 8.5.
Altogether, it can be seen that the configurable axonal delay is much higher than the expected total event communication latency $d >> l_{\text{tot}}$. Therefore the delay buffer size can be set to

$$S_{\text{dbuf}} \geq \max(d - \min(l_{\text{tot}})) \approx \max(d) \tag{7.7}$$

which is 2^{14} entries, implemented in block RAM FPGA resources. Thereby the implementation can handle any configured delay without having to drop events at the receiving side due to a full buffer.

7.5.3.2 Comparison of Values with Possible Overflow

Similar to the overflow issue in Section 7.1.3, when comparing the received events' timestamps to the current globally synchronised systime, one, both or none of these operands can overflow their binary range. Particularly, the timestamp can overflow while adding the axonal delay value at the source and the systime can overflow due to the time passed while transmitting the event. This leads



(a) Neither reception systime nor timestamp does overflow.

(b) Reception systime as well as timestamp both overflow

(c) Either systime or timestamp does overflow.

(d) Error condition where the systime overflows by more than half the timestamp binary range.

Figure 7.8: Visualisations of different overflow comparison situations between systime (`st`) and timestamp (`ts`). An event with timestamp is transferred from one systime to another (solid edges). After reception, the timestamp is compared to the current systime, whether it lies in the future (**green**, dashed edges) or in the past (**red**, dashed edges). The notation of values is combined binary (before the `'`) and hexadecimal (after the `'`). Timestamps are transmitted with limited precision, which is indicated by one MSB less on the transmission edges than at the side-labels.

to the fact that a pure comparison by value is not always sufficient to yield the correct result.

Figure 7.8 shows a visualisation of some example cases for these situations. If none (Figure 7.8a) or both (Figure 7.8b) of the operands have overflown half their binary range, the normal comparison will yield the correct result. In these cases, the Most Significant Bits (MSBs) of both operands are equal. If however exactly one of both operands has overflown half of its range (Figure 7.8c), the MSBs are different. In this case, the result of comparing the remaining bits apart from the MSB (the LSBs) has to be inverted to obtain a correct overall result.

This algorithm will however only work if neither of the operands has overflown by more than the full binary range. Otherwise, the inversion is basing on a false assumption on the actual value of the received timestamp, as shown in Figure 7.8d. In this example the too long transmission latency (or the too short timestamp) leads to a misinterpretation of the timestamp being in the (far) future, while it actually lies in the (far) past.

7.6 Configuration and Status Interfaces

As already explained in Section 3.2.5, the configuration space is split into two different systems. On the one hand there is the EXTOLL registerfile, also used by the network hardware and corresponding FPGA IP units. On the other hand there is the Omnibus system communication bus which is used by the BrainScaleS (BSS) system. The configuration and status of the spike event communication units, described above, is placed in the EXTOLL registerfile, in order to make it available for access from everywhere in the network through a natively supported special packet format (distinguished by a single bit flag in the packet-header, cf. Appendix B.2). Besides that, the EXTOLL registerfile hardware is conveniently auto-generated from a generic description together with a clear address-space documentation.
However, the BrainScaleS-2 (BSS-2) experiment flow, supported by the system software, is based on playback programs, executed by the *Playback Executor* unit on the FPGA (cf. Section 3.2.2 and Section 3.3). Therefore, a bridging unit has to be implemented from the Omnibus to the EXTOLL registerfile in order to make this configuration space available to the playback-programmed experiment flow. Figure 7.1 shows the main logic building blocks of this configuration and status communication infrastructure. At first, the Omnibus data path must be widened (cf. Section 7.6.2), as it is implemented as a 32 bit-wide signal bus while the EXTOLL registers are up to 64 bit wide. After this, the actual bridge (cf. Section 7.6.4) provides access to the local registerfile address space, as well as a global access to all the registerfiles in the network, including those of the Tourmalet cards (cf. Chapter 4).

7.6.1 The Omnibus Interface

As a basis for the description of the Omnibus data-width converter and ODFI bridge below, the relevant aspects of the Omnibus interface will be introduced here. Generally, the Omnibus communication protocol, used in the BrainScaleS hardware is a simplified version of the Open Core Protocol (OCP) specified in (OCP 2009).
Figure 7.9 shows a schematic definition of the Omnibus interface as it is used in the BSS-2 system. It mainly consists of two signal groups, one group that is driven by master units to the interface and another that is driven by slave units. An access is always initiated by a master unit requesting a command transaction (`MCmd`) to a specific address (`MAddr`) and optionally containing payload data (`MData`). The payload data can be signalled to be only partially valid on a byte level, using the `MByteEn` signal. The connected slave unit will in any case acknowledge the acceptance of the command request (`SCmdAccept`). Then, it either forwards the command, acting as another master unit down the hierarchy of the bus tree, or responds to the requested command itself by sending an `SResp` code optionally containing payload response data (`SData`). The master unit will in turn acknowledge the reception of the response by asserting `MRespAccept`.
The relevant command and response codes, used in the BSS-2 implementation at hand are listed in Listing 7.7. Command codes especially include a write (`WR`) and read (`RD`) command besides an `IDLE` code, used to signal *no command* on the bus. The slave unit will always acknowledge successful execution of a command by sending a `DVA` response, even if it is a write command. If an error occurred, like e.g. receiving an invalid command address, the slave will respond with `ERR`. An

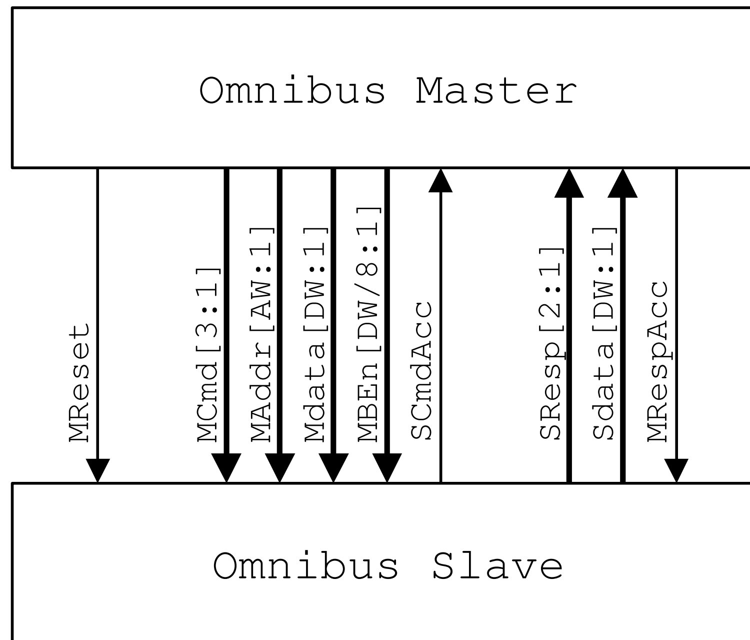


Figure 7.9: Schmatic definition of the Omnibus Interface.

```
typedef enum logic[2:0] {
  IDLE       = 3'b000,
  WR         = 3'b001,
  RD         = 3'b010
  // ...
} Ocp_cmd;

typedef enum logic[1:0] {
  NULL        = 2'b00,
  DVA         = 2'b01,
  ERR         = 2'b11
  // ...
} Ocp_resp;
```

Listing 7.7: Extract from the Omnibus OCP command and response type definitions (OCP 2009).

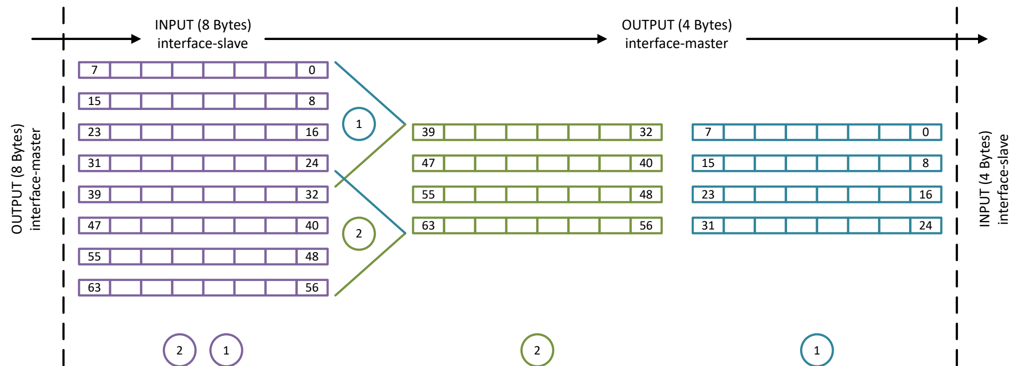
idle response bus is signalled by the `NULL` code.

Generally, a master unit is obliged to keep an active command asserted until the slave has acknowledged its reception with `SCmdAccept`. Analogously, a slave unit is obliged to keep an active response asserted until the master has acknowledged its reception with `MRespAccept`.

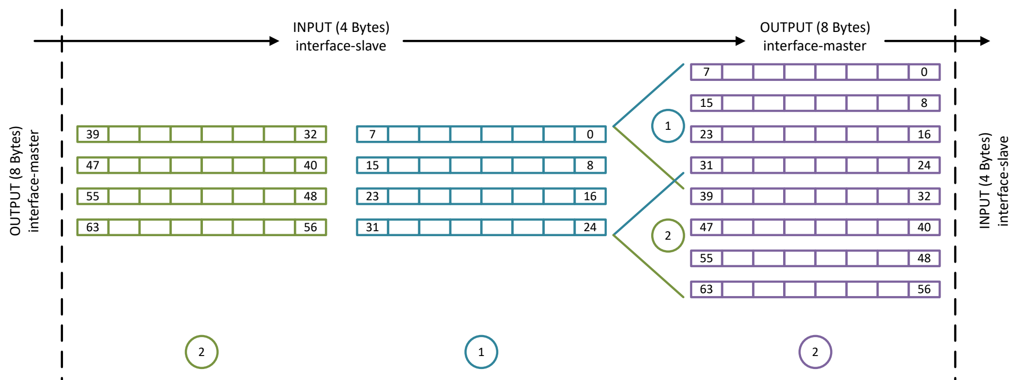
7.6.2 The Omnibus Data-Width Converter

The omnibus communication system, used in BSS-2 implements a variety of hardware units, acting as Omnibus masters and / or slaves. Existing units include examples like a `bus_switch` or a `bus_delay`, used for building and distributing an address space across a physical design area, as well as converters from and to FIFO- and RAM interfaces and a `bus_reg_target` used as registerfile target to the Omnibus address space.

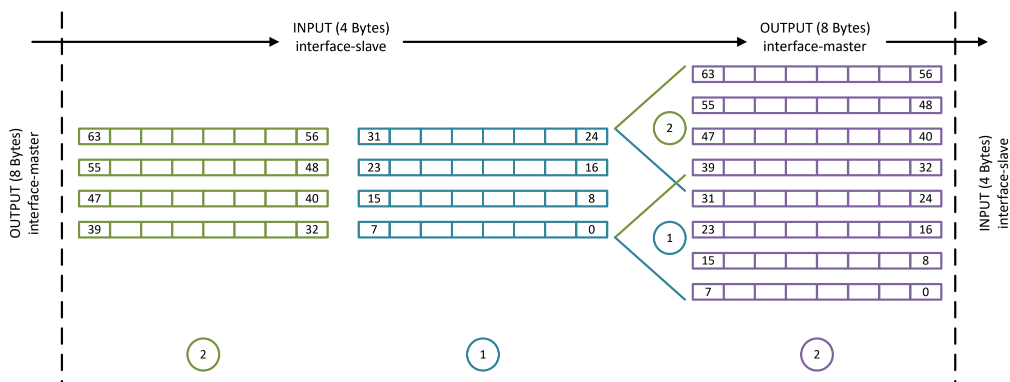
In the scope of this thesis, a `bus_data_width_converter` was implemented, which can be parametrised for different input- and output data widths which however have to be multiples of each other. It implements narrowing as well as widening of data transactions in either read or write direction.



(a) Narrowing operation with little-endian byte ordering.



(b) Widening operation with little-endian byte ordering.



(c) Widening operation with big-endian byte ordering.

Figure 7.10: Operation principle of the Omnibus data width conversion unit.

Figure 7.10 shows the operation principle of the data width converter. In case of narrowing operation (cf. Figure 7.10a), the input slave splits an incoming transaction into multiple parts according to the input to output ratio. The incoming transaction is blocked until the output master has successfully transferred all sub-transactions sequentially across the more narrow output interface. Only then does the slave acknowledge the acceptance of the input transaction by asserting `SCmdAccept`.

Alternatively, the slave could also immediately accept an incoming transaction that would then have to be buffered in an additional FIFO memory until it is fully forwarded. However, this would not improve the overall throughput, but merely free the previous pipeline stage. If this previous stage were a splitter-unit addressing distinct parts of the address space, the availability of the adjacent address space part would be improved. As the narrowing operation is not needed in the current design, this optimisation is left open for future improvement of the unit.

In the opposite case of widening operation (cf. Figure 7.10b) the input slave has to combine subsequent partial transactions to assemble the full outgoing transaction. The least significant part of an input transaction's address thereby identifies, to which position it belongs in the larger output transaction. The converter unit keeps track of the already received subtransaction positions and only issues the valid output transaction if all subtransactions have been received at the input. This information is then stored in a FIFO buffer for the later generation of responses. Furthermore, it is checked that the subtransactions arrive in ascending order with respect to their address. If these conditions are not satisfied, i.e. a subtransaction arrives with lower address than the previously accepted one, or a transaction for a different address is received before all subtransactions for the current address have been received, the output transaction is cancelled and an error-condition is stored in the FIFO to later generate appropriate responses. An error condition is also generated in case a parametrised timeout value exceeds after accepting a subtransaction, or if a received subtransaction does not match the previous one in its command type.

It should be noted that for the response path, the operation mode with respect to narrowing or widening is inverted as compared to the command path. This means that for narrowing operation in forward direction, the sub-responses have to be accumulated, while for widening operation in forward direction, the single response has to be divided back to individual responses to the previous subtransactions. Therefore, a correct set of read sub-commands to a widening converter will generate the appropriate set of responses and an error condition at the input slave of this converter unit will generate an `ERR` response to each received sub transaction, without forwarding the erroneous request through the output master interface.

Additionally to the input and output widths, the data width converter is parametrised to whether the conversion is to happen in little- or big-endian byte order. This is important, as the order to which the subtransactions are serialised or de-serialised depends on this parameter. To illustrate this, the widening operation is depicted both for little-endian (Figure 7.10b) and big-endian (Figure 7.10c) byte ordering.

For the BSS-2 FPGA implementation, the Omnibus data-width converter is needed in the parametrisation with 4 B input and 8 B output in little-endian byte order. Its purpose is to widen the Omnibus interface from the commonly 32 bit wide data bus to the 64 bit data bus of the EXTOLL ODFI registerfile. Therefore, in order to access the registerfile via the BSS-2 omnibus, two Omnibus transactions have to be issued towards subsequent addresses at 4 B granularity. These are then converted into a single 8 B transaction with also one bit less in address granularity.

7.6.3 The Registerfile Interface

Before going into the details of the bridging unit between the Omnibus configuration bus and the EXTOLL registerfile, the signal interface of the latter first has to be introduced in detail. Figure 7.11

shows a schematic definition of this interface. The master agent can request either a read- or write access by providing an `address` and asserting either the `read_en` or `write_en` signal respectively. In case of a write transaction, the master agent will also provide payload data using the `write_data` bus. The slave agent will then some time later acknowledge the completion of that transaction request by asserting the `access_complete` signal. If the transaction has failed this is most likely due to an invalid address and the slave will notify this by asserting the `invalid_-address` signal. If the transaction has however been successful, the slave will respond the obtained data in case of a read transaction.

Although all bus-widths are freely parametrisable in the given interface definition and in principle registers of any width can be implemented, the top interface's data width is always defined by the widest register in the registerfile. In the end, the implementation of the EXTOLL network protocol limits the upper limit of registers to 64 bit. As many registers in the FPGA as well as on the Tourmalet network ASIC use the full width, the top-level interface will certainly also be that wide.

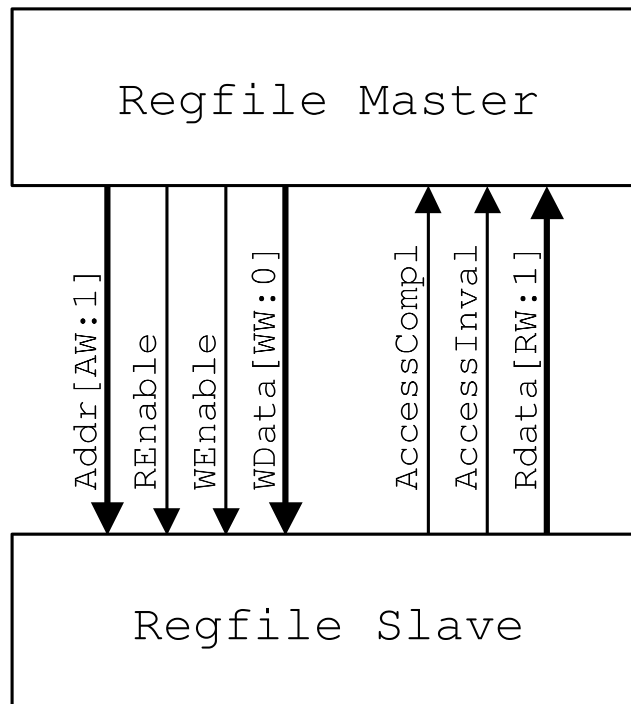


Figure 7.11: Schematic definition of the EXTOLL registerfile interface, also referred to as *Software Interface*.

7.6.4 The Omnibus to Registerfile Bridge

Now that both the formal definition of the Omnibus interface (cf. Section 7.6.1) as well as the Registerfile interface (Section 7.6.3) has been introduced, the implementation of the bridging unit between both of them can be described.

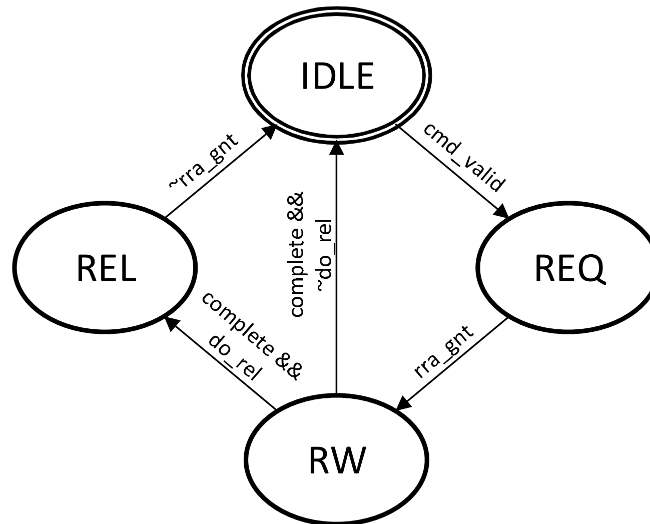


Figure 7.12: The Finite-State Machine (FSM) controlling access to the EXTOLL registerfile

7.6.4.1 Generic Registerfile Access

In general, access to the registerfile is arbitrated between the different generic registerfile access units and respectively controlled by an FSM which is shown in Figure 7.12. The unit, which is presented with an interface definition in Listing 7.8c, receives input commands which are formatted according to Listing 7.8a and responds according to the format definition in Listing 7.8b. When the input command (`cmd_in`) becomes valid (`cmd_valid`), the FSM changes from **`IDLE`** to the **`REQ`** state and requests a registerfile access permission from the arbiter (`rra_req`). When the arbiter grants the access permission (`rra_gnt`), the FSM advances to the **`RW`** state. Now the registerfile access controller unit drives the registerfile interface (`rf_access`, compare Figure 7.11) according to the transaction command. When this transaction is complete, the FSM will either advance to the state **`REL`** where the arbiter is released by asserting `rra_rel` until it de-asserts the `rra_gnt` signal, or return directly to **`IDLE`** while keeping the granted access right. The latter mode is especially useful for atomic read-modify-write accesses and is signalled by the `do_release` field in the input command.

7.6.4.2 Omnibus Registerfile Access

The actual bridging unit from the Omnibus to the Registerfile implements another FSM controlling the generic Registerfile access unit based on incoming Omnibus transactions, which is depicted in Figure 7.13. This unit operates in two different modes for accessing the local Registerfile or a remote Registerfile on another node in the network. Which mode is to be executed for an incoming Omnibus transaction, is decided by the state of a configuration register, also connected to the Omnibus system. This register configures the 16 bit target node-id for the Registerfile access to be generated from incoming Omnibus transactions. If this configuration register is marked invalid by configuration to a dedicated `valid` bit or contains the node-id of the local FPGA, the access is executed to the local Registerfile (the **purple states** in Figure 7.13). Otherwise, an access command is inserted into a FIFO queue towards the NHTL to be sent across the network to and to be responded by the

```
typedef struct packed {
  logic [AW:1] address;
  logic [WW:1] data;
  logic is_read;
  logic do_release;
} rf_cmd_t;
```

(a) Command format description.

```
typedef struct packed {
  logic [RW:1] data;
  logic err_invalid_addr;
} rf_rsp_t;
```

(b) Response format description.

```
module generic_rf_access (
  input logic clk, res_n,

  input rf_cmd_t cmd_in,
  input logic cmd_valid,

  output rf_rsp_t rsp_out,
  output logic complete,

  output logic rra_req,
  input logic rra_gnt,
  output logic rra_rel,

  rf_if.master rf_access
);
```

(c) Module interface definition. `rf_if` references the registerfile interface, as defined in Figure 7.11.

Listing 7.8: Systemverilog module interface descriptions for the generic Registerfile access unit.

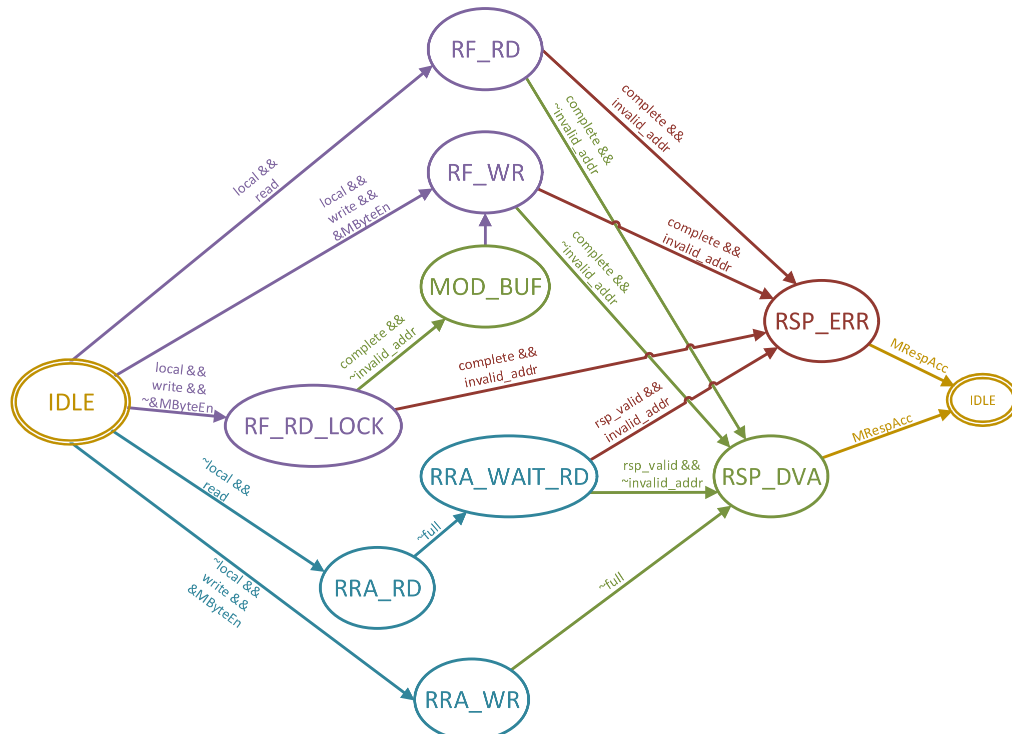


Figure 7.13: The Finite-State Machine (FSM) converting Omnibus transactions into Registerfile access transactions.

target node (the **blue states** in Figure 7.13). The potential response (in case of a read transaction) is returned by the NHTL through another FIFO queue. The remote node will in between access its own local Registerfile with the received transaction like for the usual Registerfile access transactions initiated by the host software, as described in (Thommes 2018).

In both cases (local or remote access), the FSM distinguishes between read- and write transactions. In case of local write access **RF_WR**, a special atomic read-modify-write access mode is triggered by only partially valid `MByteEn` signals. In that case, the Registerfile address is first read while keeping the acquired access permission grant from the arbiter (state **RF_RD_LOCK**). The register content is then modified at the byte positions, indicated by the `MByteEn` signal in the state **MOD_BUF**. After this, the FSM switches to the normal local write state **RF_WR** feeding the *Generic Registerfile Access* unit with the write transaction and using the same grant that was locked at the read transaction before. Thereby it is ensured that both access transactions are executed atomically, without another Registerfile master reading or modifying the respective address in between. After the write access has succeeded, the access permission is released back to the arbiter.

In case of a remote Registerfile access this read-modify-write transaction mode is not possible due to restrictions in the EXTOLL network protocol. Also only read transactions **RRA_RD** are notified back to the source node with either the content response or an error notification in case of an invalid address request. Therefore the FSM also only has to enter a waiting state (**RRA_WAIT_RD**) for remote read accesses. Remote write accesses (**RRA_WR**) will immediately return after issuing their command into the FIFO queue towards the NHTL.

The *Omnibus Registerfile Access* unit will respond each Omnibus transaction either with **RSP_DVA** in case of successfully executed Registerfile access transactions or **RSP_ERR** in case of invalid Registerfile address requests. However it has to be noted that for remote write accesses, due to the lack of a response or error notification message, invalid address requests will not be notified, but instead (wrongly) acknowledged by an **RSP_DVA** response.

7.6.5 Registerfile Access Arbitration

Registerfile accesses have to be arbitrated and multiplexed onto the interface of the top Registerfile unit, as multiple agents, like e.g. the NHTL and the Omnibus bridge act upon the Registerfile. If these agents were not arbitrated, their access transactions would most likely interfere on the Registerfile bus structure and corrupt each other. Also the arbitration enables atomic read-modify-write access patterns without another master agent interfering with the same register address (cf. Section 7.6.4.1 and Section 7.6.4.2). As these master agents might operate in different clock domains relative to each other and to the top level Registerfile unit, the arbitration signals (`request`, `grant` and `release`), as well as the Registerfile interfaces themselves have to be synchronised into the target clock domain of the Registerfile top level and back to the respective requesting clock. A schematic block diagram of the Registerfile access arbitration multiplexer is shown in Figure 7.14. The unit can be parametrised for the number of input interfaces competing for exclusive access to the Registerfile. The arbitration itself is done using a Round Robin scheduler (cf. Section 2.3.3.1).

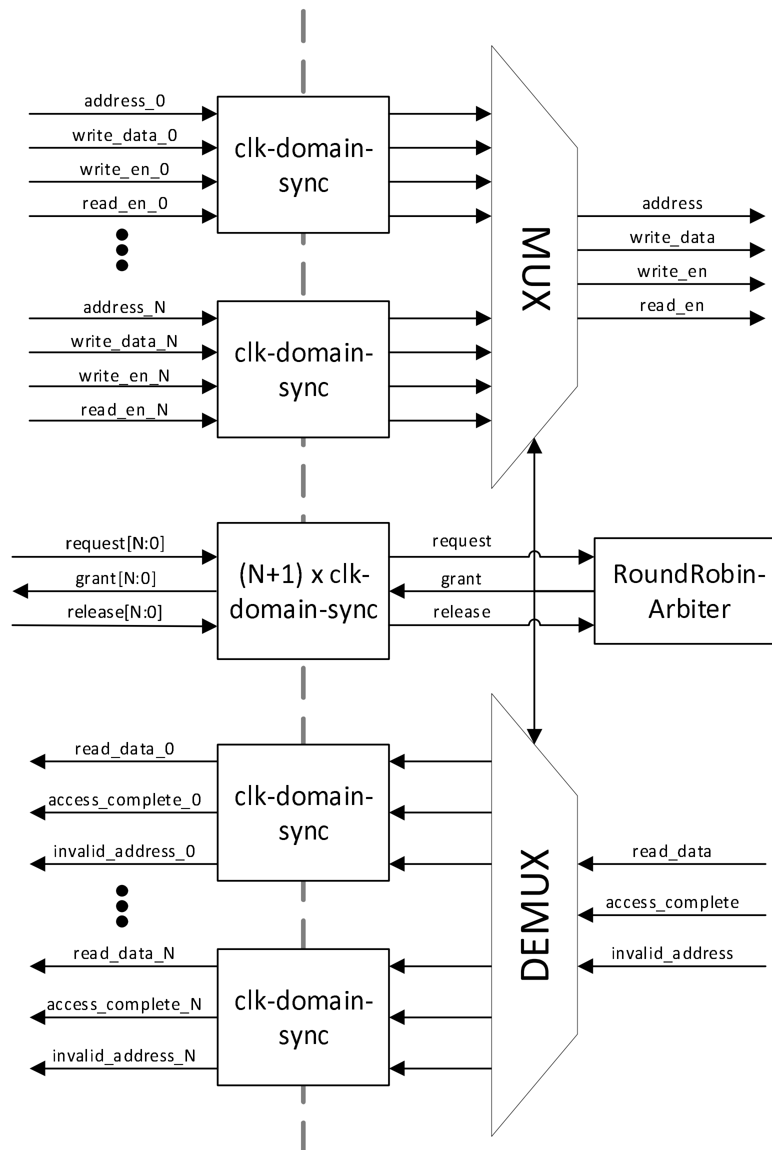


Figure 7.14: Block diagram of the Registerfile Access Arbitration Multiplexer. Input slave interfaces may reside in different clock domains relative to each other and relative to the output master interface.

7.7 Clock Domain Signal Synchronisation

Generally, the need for signal synchronisation between different clock domains arises from the setup- and hold time constraints of flip-flop circuits. A usual flip-flop samples and stores the input data at the rising edge of a clock signal. In order for the stored value to be stable, the input signal may not change during a certain time interval before the rising clock edge, which is called the setup-time. Still, the output signal of the flip-flop will only become stable if the input is kept stable for another certain time interval after the clock edge, which is called the hold-time. Normally, the place and route implementation compiler ensures that these constraints are met and the logic and wire delays in between two design registers do not exceed their allowed margin at an FPGA or ASIC implementation. However, when two connected registers are located in different clock domains and

the respective clock signals' frequencies are asynchronous to each other, it is generally not possible to fulfil these timing constraints at all times. Inevitable setup- and hold time violations will cause the output signal of the second flip-flop to go in a metastable state (cf. Figure 7.15).

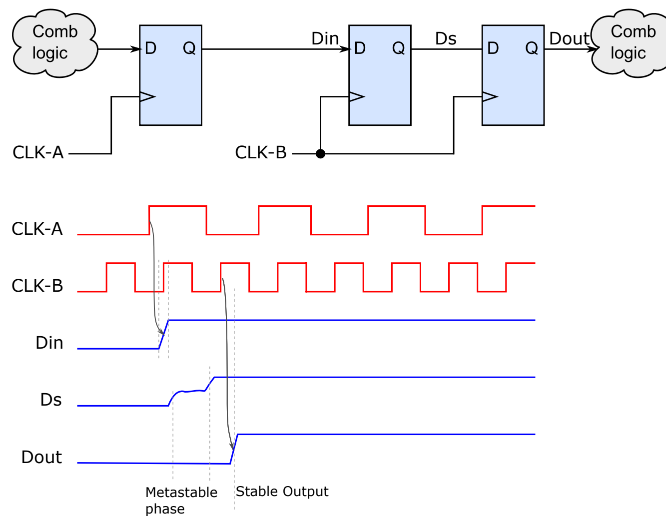


Figure 7.15: Metastability in digital circuits: Simplified schematic how a data line can cross different (asynchronous) clock domains (`CLK-A` and `CLK-B`) to minimize the risk of metastability with a double buffer D-Flipflop chain. Figure taken from (WikimediaUser 2015)

How long this state will last, cannot be determined with certainty, but the probability to stay in the metastable state will decrease exponentially with time. Depending on the time constant of this probability, the metastability can be filtered out by chaining one ore more additional flip-flops. The probability that the metastable state of the first flip-flop will cause another timing violation at the second or third flip-flop will then be very near zero. However, this simple synchronisation strategy is only suitable for logically independent control signals, as it can take one or multiple cycles until the synchronised signal resembles the value of the original signal. This may skew the individual bits of a data bus in time and thereby corrupt the transported data. Furthermore, this only works for cases where the source clock is slower than the target clock, as otherwise, the synchronised signal might miss fast pulses on the source signal.

The problem of synchronising a data bus across a clock domain crossing can be solved on different ways. One way is to implement a synchronised handshake mechanism where the target clock domain only latches data from the source register, when a valid-signal has been synchronised across the border. The source clock domain on the other hand does only update its data after an acknowledge signal has been synchronised back from the target domain, indicating that the data has been read.

Another possibility is to use an asynchronous FIFO where data is inserted from the source clock domain and extracted in the target clock domain. Thereby, the write- and read pointers have to be synchronised between both domains. A one bit wide FIFO can also be used to synchronise a pulsing signal from a fast clock to a slow clock. Alternatively, one can count the number of clock cycles, the signal is asserted in the source domain and synchronise that number to the target domain in order to assert the output signal for the exact same number of cycles there. The synchronisation of the counters or read- and write pointers between the clock domains can again be handled by handshaking.

In the event communication design, described in the previous sections, clock domain synchronisation modules (handshake- and count synchronisation) are used that have been provided by Benjamin Kalisch at the Extoll GmbH in the year 2008, as well as a synchroniser module for the Registerfile interface, based on the previously mentioned modules and provided by Benjamin Geib at the CAG in 2011.

7.8 Design Parametrisation

The event communication design, described in the previous Sections of this Chapter is mainly parametrised by two parameters that are the number of accumulation *Buckets B* and the number of parallel event data paths, called *Splits S*.
As indicated in Figure 7.1, the parameter *S* thereby especially determines the number of lookup tables for destination mapping (cf. Section 7.3.2) as well as the number of delay buffers at the receiving side (cf. Section 7.5.3). Those units, as well as the *Accumulation Buckets* are configured and provide status information through their respective sublayer of the Registerfile. Therefore the overall Registerfile address space, as well as the code block for instantiating and connecting the respectively generated sub-Registerfile units depends on the values of these parameters. As the Registerfile code is generated from a generic TCL source description (cf. Section 3.2.5.2 and (Computer Architecture Group 2018)), this description actually generates itself based upon these parameters using TCL loops and conditional branches. Additionally, the Registerfile instantiation blocks have to be programmatically inserted into the respective functional unit's Register Transfer Language (RTL) source code. This is handled by short postprocessing scripts (implemented in AWK language) that are executed after the Registerfile generator (cf. Section 3.2.5.2) has finished generating the Registerfile RTL description. The functional units' RTL code itself is parametrised using the Systemverilog language construct of interface arrays (for details please refer to (SystemVerilog 2004)).
To ensure that all code generating instances (the Registerfile generator, the postprocessing scripts) and the functional units' RTL source code itself work on the same parameter values, these values are exported to the environment by the central Makefile script before starting the Registerfile generator scripts and the FPGA implementation toolchain.
Similarly, the numbers of implemented EXTOLL barrier- and interrupt units (cf. Section 4.1.2) are parametrised in the FPGA design. While it does not make sense to implement more than 16 barrier- and 4 interrupt units, which are the numbers provided by the EXTOLL Tourmalet network ASIC, it might be desirable to save precious FPGA resources, as this amount of barrier- and interrupt units will not be needed by application purpose of synchronising a single global systime across the FPGAs taking part in a neuromorphic multi-chip experiment. However, as will be shown later (cf. Section 8.2), implementing the full number of barrier- and interrupt units does not pose a recourse problem to the design.

8 Commissioning

In the last Chapter, the implemented event communication architecture for the BSS-2 neuromorphic computing system has been described and explained in detail. This Chapter will now elaborate on the measures that have been taken in order to verify (cf. Section 8.1), commission (cf. Sections 8.2 to 8.4) and characterise (cf. Sections 8.5 to 8.7) the developed communication architecture in the context of the BSS-2 system.
The last Section finally demonstrates the successful operation of a biologically motivated neuromorphic network model (the Synfire Chain model, cf. Kremkow et al. 2010) across two BSS-2 ASICs with the developed and implemented communication architecture.

8.1 Simulation and Verification

In order to verify the design with respect to data integrity and functional correctness of the configuration bus access, two simulation testbenches have been implemented. These testbenches will be presented in the following subsections.

8.1.1 Event Communication

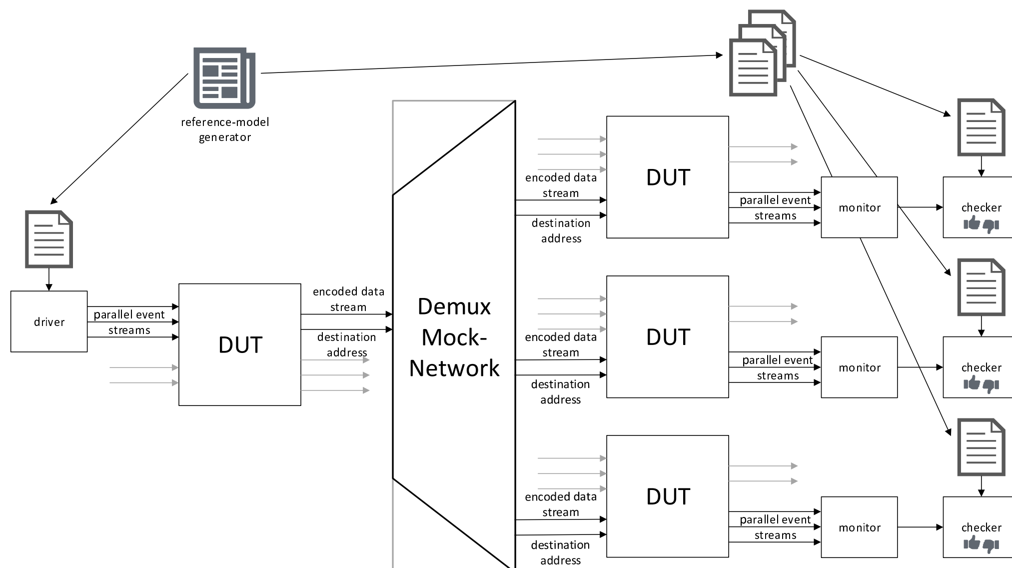


Figure 8.1: Schematic block diagram of the event communication testbench. One Design Under Test (DUT) instance is used exclusively as sending unit and is parametrised with a number of *Buckets B* and parallel data paths (*Splits*) *S*. The network, connecting the single sending unit to multiple receiving units is mocked using a demultiplexer. The receiving units are further DUT instances and only the receiving part is used.

The first testbench exclusively simulates and tests the event communication architecture and is visualised as a schematic block diagram in Figure 8.1. For this purpose, the event communication DUT is instantiated multiple times and the instances are connected through a mock network, modelled by a demultiplexer unit. A single DUT is used only for the sending side of the design, while the other DUT instances are used as receiving units. The sending DUT receives parallel event streams according to the parametrised number S of data path *Splits* and distributes them across the parametrised number B of *Accumulation Buckets*. These buckets are configured with ascending destination Node IDs (NDIDs) resembling the index under which the respective receiving unit is connected to the demultiplexer mock-network. Finally, the encoded data streams respectively arriving at the receiving DUTs are decoded back to event streams.

The event data streams, used for the test are randomly generated and stored to files by a Python script. This data generating script also implements the reference model and further writes the respectively expected output event streams at the receiving units to files. Furthermore, it generates the configuration data for the destination mapping lookup tables.

The testbench's main execution procedure will at first configure the sending DUT, using an instance of the *Generic Registerfile Access* unit (cf. Section 7.6.4.1) Subsequently, the testbench will execute the stored input event streams into the sending DUT. The receiving DUTs are simultaneously monitored and their output event streams are compared to the previously generated expectation data. This testbench design verifies that the previously described event communication architecture correctly maps the input events onto the right bucket and output event label. Furthermore it is verified that the UT encoding- and decoding units are correctly parametrised and that the implemented encoding scheme is able to correctly transfer the event information. However, all the timestamp handling aspects of the design are not tested with this testbench. This will be tested later in the real world system (cf. Section 8.7).

8.1.2 NHTL and Registerfile Access

The second testbench, employed for verification of the design at hand is the one also used in (Thommes 2018). Figure 8.2 shows this testbench in the form of a schematic block diagram.

It is based on the Universal Verification Methodology (UVM) and has been extended to also include the event communication ports of the *NHTL unit* (cf. Section 7.4) and the Registerfile access via an *Omnibus Data-Width-Converter* (cf. Section 7.6.2) and *Omnibus Registerfile-Access-Bridge* (cf. Section 7.6.4). Additionally, the existing interface Universal Verification Component (UVC) at the Application-Layer (AL) interface has been modified to be able to passivate the driver components. This is useful, as the existing DUV has been extended by the AL-Test interface unit which will be presented in Section 8.1.5.4. This interface testing unit can be configured to generate patterns to the AL interface or loopback the received transactions. In these cases, when the AL-Test unit is not configured to bypass mode, the testbench drivers have to be deactivated, as they are not needed. For this purpose, the respective components are parametrised with a switch bit that is to be set at the build phase of the simulation.

The Scoreboard (SCB) now additionally checks that all data transactions that are monitored at the `evtcomm2nhtl` port is also seen at the `hbp2np` port in the correct packet format. In the opposite direction, the SCB checks that the content of packets arriving with event communication data at the

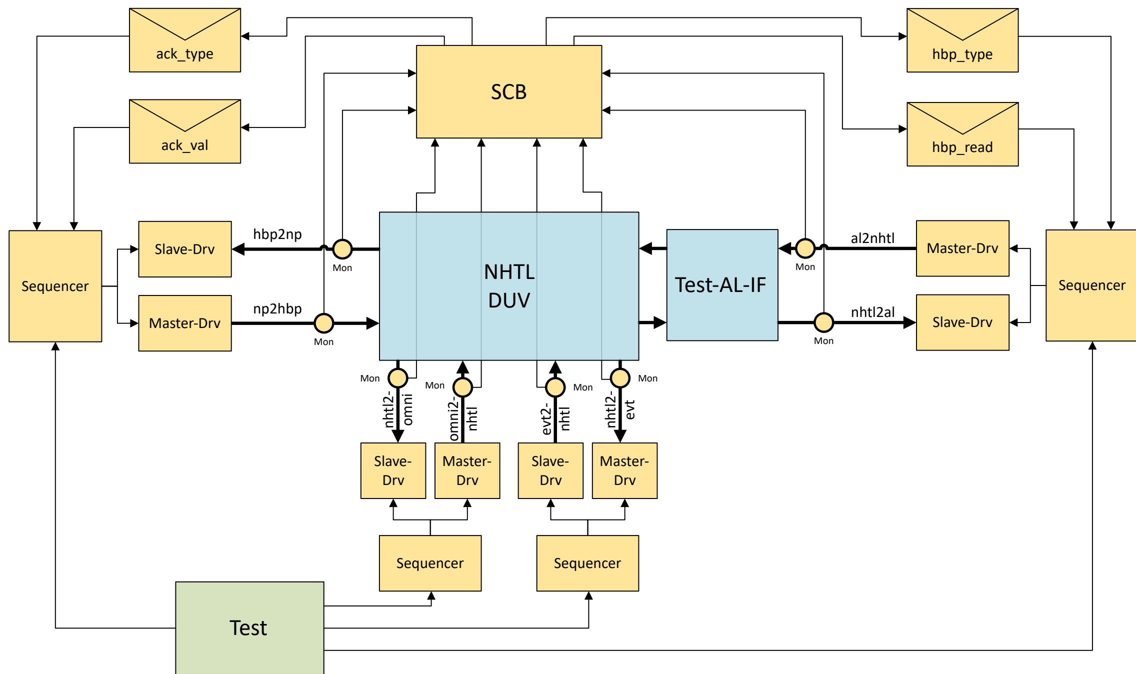


Figure 8.2: Schematic block diagram of the extended NHTL testbench, modified from (Thommes 2018). Test sequences are driven through the respective interface sequencer and into the Design under Verification (DUV). Monitors scan the interfaces and report transactions to the Scoreboard (SCB) where they are buffered and compared to transactions from other interfaces.

`np2nhtl` port will also arrive at the `nhtl2evtcomm` port output.

For the Omnibus Driver and Sequencer components, an existing interface UVC agent was reused that is also used for verification of the BSS-2 ASIC. This is one of the key advantages of the UVM methodology, as existing interface UVCs can easily be ported between different testbenches and module UVCs.

In the scope of this thesis, several additional test sequences were added to the verification environment as compared to (Thommes 2018). These additional tests will be shortly summarised in the following paragraphs.

8.1.2.1 tx_evt_data_test

This test inserts random event data transactions at the respective NHTL interface port. The SCB then checks that these transactions are correctly transformed into network packets in agreement with the protocol requirements the EXTOLL NP.

8.1.2.2 rx_evt_data_test

This test inserts EXTOLL network packets into the NHTL's NP interface port. These packets are generated, to contain random event data that should be forwarded to the receiving event communication interface port by the NHTL unit. The SCB checks that the data is correctly forwarded and received at the interface.

8.1.2.3 put_random_test

The **put_random_test** already existed in the original test library in. It was however extended to also include the random generation of event communication packets into the NP interface and event-data transactions in the opposite direction, alongside the other packet types. This summarises the functionality of the **tx_evt_data_test** (Section 8.1.2.1) and the **rx_evt_data_test** (Section 8.1.2.2) together with the existing functionality, described in (Thommes 2018). Transactions of any type and at any of the NHTL's interfaces are generated randomly and independently of each other. This is a major stress test to the NHTL as all data paths are used simultaneously.

8.1.2.4 rra_omnibus_test

This test performs Registerfile write- and read transactions via the two access paths available to the design. It first writes some values to the Registerfile via the Omnibus Bridge (cf. Section 7.6.2 and Section 7.6.4) and subsequently reads them back via the same Omnibus path and via the native EXTOLL path. The results read back from the register are both compared to the value written via the Omnibus. Afterwards, the same procedure is repeated, but this time the values are first written via the EXTOLL access path and read back on both paths.

8.1.2.5 test_alif

This test sequence configures the AL-test unit (cf. Section 8.1.5.4) through its Registerfile to generate test transactions with a fixed pattern. The test then polls the status register of the test unit for completion of the test pattern generation. When all test patterns have been generated, the test repeats the procedure for multiple iterations. As with this *internal* test data generation mode the interface driver component is not needed, it is disabled at the simulation build phase of the testbench environment. The SCB will again check that all transactions are correctly packed and transmitted through the NP interface.

8.1.2.6 loopback_test

Similar to the previously described **test_alif** sequence (Section 8.1.2.5), this test sequence configures the AL-test unit after deactivating the respective UVC drivers. However, this time, the test unit is configured for loopback operation and test input is inserted in form of NP packets through the NP interface driver. There are variants of this test sequence for all the AL interface types.

8.1.3 Continuous Integration

The Electronic Visions group uses Continuous Integration (CI) to continuously build and verify the software- and design code. The CI system (Jenkins) can trigger build processes on a timed schedule, e.g. every night or once a week, or (additionally) at every code change that is committed to a code review server (Gerrit). (Müller, Mauch, et al. 2020; Müller, S. Schmitt, et al. 2020) It is also possible to trigger CI jobs for repositories that depend on a code-change in another repository, thereby excluding passive dependency bug insertion. This is used in the scope of this thesis to run the verification simulations, described above at every code change and regularly scheduled. This

approach drastically reduces the probability of introducing major bugs into the design through code changes. Additionally, the scheduled regular simulation execution also implements a test regression to detect bugs that only occur in rare corner cases.

8.1.4 Hardware-Software Co-Simulation

In addition to the functional verification, driven by unit test simulations like the ones described above, the `HXFPGA_CORE` partition is simulated as a whole using a system testbench.

This system testbench is directly driven through the same software stack as the live system (cf. Section 8.4). The connection between the HDL design testbench and the software stack is thereby established through a Systemverilog DPI (IEEE 2018) interface layer of the software stack. This interface layer also directly controls the hardware simulator. The DPI interface allows to call C-functions from a SystemVerilog task or function and vice versa. Thereby, the software can transfer transactions to the simulation by calling a SystemVerilog task and receive transactions from the simulation by having an according function called by the simulation process.

This so-called hardware-software co-simulation is used to run a large number of software defined test cases that can be specific to a particular version of the BrainScaleS hardware or generally applicable to all hardware versions. These tests can address the FPGA design as well as (a part of) the ASIC, also simulating the communication links between both parts of the system. Mainly, this approach enables development of the required changes to the software stack in parallel to the development of a respectively new hardware version. Also it facilitates the easy development of high level test cases for verification of new FPGA and software features. Thereby, possible bugs can be directly investigated on the simulator output in a graphical waveform view.

Unfortunately this approach is not easily applicable to a neuromorphic multi-chip environment, relying on integral features of the EXTOLL network, as it would require a full behavioural model of the Tourmalet network ASIC. A possible solution would be to exclude the details of the EXTOLL network and only simulate the remainder of the BrainScaleS FPGA and -ASIC, at least additionally including the NHTL. This will however require to adapt the libRMA (cf. Section 4.4) to create DPI calls instead of invoking the EXTOLL kernel-driver calls for the actual hardware access. This adaptation of the libRMA has been done by Leonard Henger at the Computer Architecture Group, but has not been tested in the BSS-2 simulation environment yet (cf. Section 9.2 on page 195).

However, the benefit of this additional system simulation environment is also open to discussion, as the NHTL is already verified, having the unit testbench described in Section 8.1.2. Furthermore, the event communication between multiple FPGAs would not be simulatable to a higher extent, than already done with the unit testbench, described in Section 8.1.1. Especially, the transmission latencies and protocol constraints of the real network cannot be simulated without having a detailed simulation model of the EXTOLL hardware. Also the timestamp synchronisation that entirely relies on the EXTOLL barrier and interrupt cannot be simulated without detailed model information. The only benefit of this simulation interface would be testing the Neuromorphic Hardware Transaction Layer via Extoll (NHTL-Extoll) software library against a simulated FPGA design, which also can and has been done in the real hardware (cf. Section 8.1.5.4, Section 8.4.2 and Section 8.4.7).

8.1.5 Design for Test and Live Debugging

As a simulation model of the EXTOLL hardware has not been available in the course of this thesis work, integration tests of the described FPGA design aspects have to be executed on the real hardware system. In order to facilitate this need for further verification, the design has been built including multiple methods of *Design for Test*. This term generally describes the approach of designing a system to directly include tools and handles that will later help verifying and testing the manufactured ASIC or programmed FPGA, either using special hardware test equipment or directly in the target system. In the case of a manufactured ASIC this is usually not suited to find vital design flaws in the first place, as the process of manufacturing the ASIC is quite expensive and time consuming. However, it can be very useful and even necessary for finding and sorting out chips with manufacturing defects or variations out of the specified range of tolerance. In the case of FPGA designs however, the cost of manufacturing is not existant as the only penalty is the time needed to compile the bitstream file and program it to the FPGA hardware. The former is mostly a matter at the order of one hour and the latter on the order of minutes. Therefore, one can resort to hardware integration tests also for (high level) verification purposes in the realm of FPGA design development.

8.1.5.1 Scan Chains

Generally, when employing *Design for Test* methods on a hardware design, auxiliary structures are added to the design, allowing to access and manipulate specific state registers or logic signals. An example for such structures can be a scan chain running through all the register bits or the IO-ring at the border of an ASIC. An Example for such a scan chain structure is the JTAG protocol (IEEE 2013) offering a method to test and debug integrated circuits by injecting instructions and moving data into and out of the circuit through a shift register architecture. In the BSS-2 system, JTAG is used for debugging, as well as for the initialisation and bring-up of the high speed serial links between the FPGA and the respective ASIC (Hartel 2016; Rettig 2019b).

8.1.5.2 ChipScope Debugging

Insertion of debug structures, like e.g. scan chains, into a design has been automated in most available tool-chains. Especially for FPGA design, the Xilinx® Vivado® tool offers an automated method to insert Integrated Logic Analyzer (ILA) cores, which are connected to arbitrary signals in the design (AMD 2023; Arshak et al. 2006; Xilinx 2011). Signals which are to be observed through an ILA core are distinctively marked in the HDL source code and afterwards selected in the Vivado® tool for the actual implementation of an ILA core. The tool will implement one ILA core per involved clock domain.

These debug cores are accessible through a JTAG port that can be connected from the FPGA PCB to a dedicated logic analyser device, which in turn can be connected to a running Vivado® software instance via network or USB. The software tool then allows to program arbitrary triggers to the ILA cores. When these triggers assert true during operation, based on the involved design signals, the ILA core records some preconfigured (at the time of implementation) number of clock cycles. For the most cases, the default setting of 1024 clock cycles will be sufficient. The recorded waveform data is sent back to the software tool, where it is displayed to the user for the actual debugging.

This method was also used extensively during this work for debugging the FPGA design. However, it is often quite tedious to have to synthesise, place and route a new bitstream file every time one wants to add or change some design signals for the ILA cores. It is therefore helpful to generously select those signals that carry the most information about the state of the design part to be debugged. In case of network packet communications, managed by the NHTL, this would be the NP and AL interface signals, as well as the interface towards the `SPIKE_COMM` partition.
Another problem with this approach is that signals from the precompiled partition netlists cannot easily be marked for consideration at the ILA core. Instead signals from these design areas have to be manually routed through the design hierarchy up to the top level module of the respective partition. As this is even more tedious than simply waiting for a new bitstream file from the implementation compiler, other methods should be employed for debugging these areas.

8.1.5.3 Counter Registers

Another method for debugging a running design is to integrate counter registers for certain anticipated error conditions as well as performance counters for anticipated key events. Examples for error conditions could dropped transactions due to full buffers or other reasons, or detected protocol errors in a network packet. Examples for useful performance counter, e.g. in the NHTL unit, are numbers of sent and received packets of specific types. These registers can be monitored during operation of the design and checked for unexpected counts.
A great advantage of this method is that it is also used heavily in the EXTOLL Tourmalet ASIC where they are globally accessible in the network trough the Registerfile. This poses the only viable method to debug the behaviour of the network cards. Of course this method is only useful, if the network is still operational.

8.1.5.4 The AL-Test Interface

In order to test the NHTL unit in conjunction with the newly introduced software layer (cf. Section 8.4.1) for host communication through the EXTOLL network, an additional testing unit has been included between the NHTL unit and the `HXFPGA_CORE` partition. This so-called AL-Test-Interface has already been mentioned in the context of the respective unit testbench in Section 8.1.2. This additional hardware design unit is necessary, as it is not easily possible to simulate the EXTOLL network communication with a co-simulation setup, as explained in Section 8.1.4.
A schematic block diagram of this testing unit is shown in Figure 8.3. Incoming transaction from the NHTL can either be forwarded transparently to the *Playback Executor* or looped back to the NHTL through a FIFO buffer. Alternatively, there is a configurable test pattern generator, driving the AL-write interface towards the NHTL. In case of loopback or generated test pattern operation, the *Playback Executor* will not be driven, as its responses would not be received by the host.
The pattern generator can be configured to send up to 256 64 bit Quad Words (QWs) with a configurable pause time of up to 256 clock cycles in between. The content of these data QWs can be programmed to a fixed value or automatically increment from a start value. The generation of these test patterns will start when a trigger bit is written to the respective control register. When the pattern has been completely executed, this bit will automatically be reset by the unit.

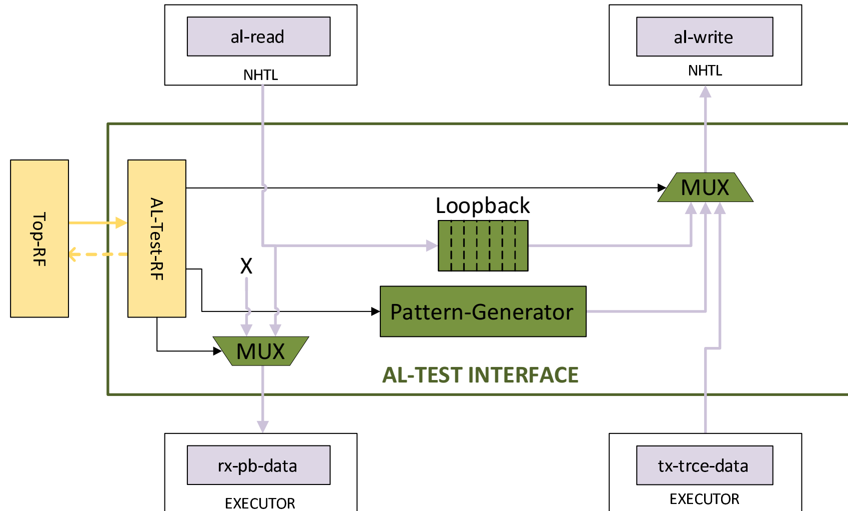


Figure 8.3: Schematic block diagram of the testing unit for the AL interface.

For debug purposes, the testing unit is equipped with a number of counter registers (cf. Section 8.1.5.3) counting the number of transactions of specific types (the AL types, as described in (Thommes 2018)) received and sent across the AL interface.

8.1.5.5 Signal Multiplexing onto GPIO Pins

Last but not least, another common debugging method is to connect signals from all over the design to a limited number of GPIO pins, using a multiplexer tree throughout the design that can be configured using control registers.
This method has been used in the scope of this work especially for measurements for which an external oscilloscope is required. This has been the case for precise latency measurements for the transmission of spike event communication packets from FPGA to FPGA (cf. Section 8.5), as well as measuring the systime synchronisation jitter across multiple FPGAs (cf. Section 8.6.3).

8.2 Physical FPGA implementation

In Section 3.2 the overall FPGA design for the BSS-2 system has been briefly described, while in Chapter 7 the implemented event communication architecture was presented in detail.
This FPGA design is compiled to a bitstream file for loading on the Kintex®-7 FPGA boards that are used throughout the BrainScaleS systems. As these are Xilinx FPGAs, the synthesis and implementation flow is mainly handled by the proprietary Xilinx Vivado® software tool. However, the BrainScaleS hardware design units, used throughout the FPGA design, especially the UT, employ IP units from the Synopsys DesignWare® Library. As these IP units come in encrypted source files that cannot be easily processed by Xilinx Vivado®, the main parts of the design have been encapsulated in a sub top-level (cf. the `HXFPGA_CORE` box in Figure 3.4). This is separately processed by the Synopsys Synplify® syntheses compiler. Finally, the synthesised netlist of this sub top-level can be imported by the Xilinx Vivado® software and instantiated into the overall design.

The event communication architecture has been developed and implemented in parallel to the existing Ethernet based single-chip design (cf. Rettig 2019a). As the evolvement of the existing design continued with general improvements, the requirement was, to always build upon these general improvements and to have the overall design mismatch between the newly developed spike event communication design and the existing one as low as possible. Therefore, the spike event communication architecture was decided to be implemented in another sub top-level which is then additionally instantiated into a clone of the existing design (cf. the `SPIKE_COMM` partition in Figure 3.4). Again it has to be pre-synthesised using the Synopsys Synplify® compiler, as it makes use of the UT units, developed by (Karasenko 2020) (cf. Section 7.3.3 and Section 7.5.1), which in turn use the DesignWare® Library.

As the *Event Switch* unit has to be inserted in the main event path of the existing design, this made it necessary, to also modify the existing `HXFPGA_CORE`. However, these changes are backwards compatible, i.e. when the event communication sub top-level is not instantiated, the interface signals connecting to the *Event Switch* can be left unconnected (in case of outputs) or passivated (in case of inputs). The design can fully operate in the existing Ethernet-based single-chip operation mode without depending on the event communication block.

8.2.1 FPGA Primitive Usage

8.2.1.1 Scaling the Event Communication Architecture

Synthesis of the `SPIKE_COMM` partition (cf. Figure 3.4) has been run multiple times for different values of the design parameters S and B, introduced in Section 7.8. Thereby, the the number of parallel data paths (*Splits*) S was swept in a range of $[2,4]$ and the number of accumulation buckets B was swept in a range of $[1,32]$. For each run, the synthesis compiler reports the number of used FPGA resource primitives.

The results of these synthesis runs for the event communication architecture, including the units for *Event Transmission* (Section 7.3) and *Event Reception* (Section 7.5) are shown in Figure 8.4. It can be seen from these plots that the usage of resource primitives rises linearly with the parameters B and S. A rise in the slope of the fitted linear functions can be explained by the fact that the buckets themselves are parametrised by the number of *Splits* S, regarding the number of input ports to the bucket. Thereby, the number of primitives scales with the product of both parameters B and S. This effect can be clearly observed for the number of required Lookup Tables (LUTs) (Figure 8.4a) and Digital Signal Processors (DSPs) (Figure 8.4d), but slightly also for the number of Flip Flops (FFs). For the LUTs, this can be explained by the amount of bucket-wise required encoding logic for the UT alphabet, described in Listing 7.6, which rises with the number of input ports, i.e. *Splits* S. For the number of FFs, this effect arises from the input register buffers, while the usage of DSPs is mainly due to the status and error counters in the Registerfile. In this case, each bucket contains one event counter per input port and one for the number of sent packets at the output.

The buffer size of a bucket does not depend on the number of input ports, but rather on the maximum packet size constraint of the network. Therefore, the slope for the number of used Block-RAMs (BRAMs) does not change with the parameter S and the number of BRAMs does not depend on the product of both parameters. Instead, each *Bucket* and *Split* adds a constant, but significant amount

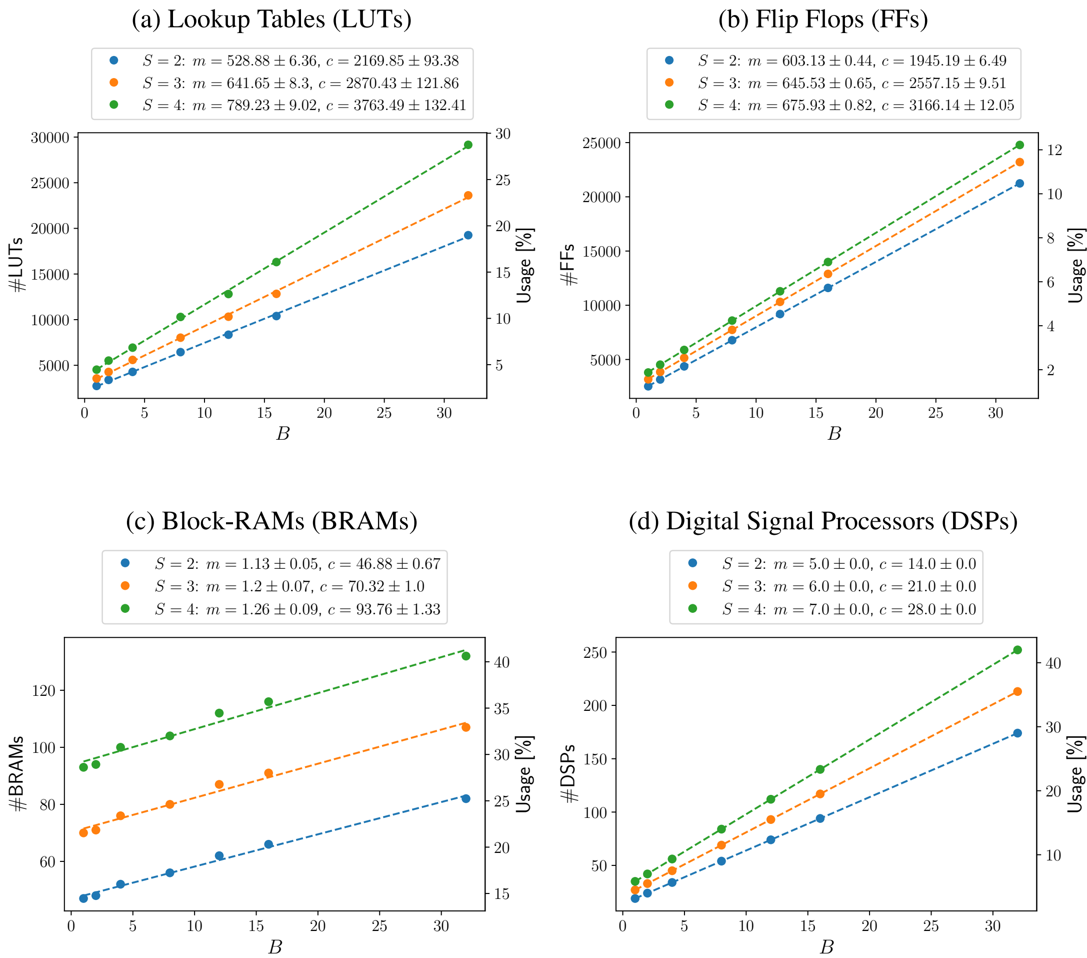


Figure 8.4: Absolute and relative usage of FPGA resource primitives for the event communication architecture with different design parameters. The data points have been fitted to a linear function $f(x) = m * x + c$. Fitted parameter values are given in the legends.

of required buffer space, as each bucket implements a packet buffer and each split implements a destination-mapping lookup table on the sending side, as well as a delay buffer on the receiving side.

Based on these numbers and on the fact that the serial links from the chip on average transport two events per clock cycle with bursts of three events per clock cycle (cf. Chapter 7 on page 109), it was decided to implement two parallel data paths ($S = 2$), rather than three. Thereby, the design can always handle the average event rate without introducing a throughput bottleneck. The bursts of three events per clock cycle are handled by compressing the triple-events into multiple double-events (cf. Section 7.1.2).

8.2.1.2 Evaluating the Overall Design

The overall primitive usage of the design is reported by Vivado® and presented in Figure 8.5. It can be seen that in total numbers the design is using quite a significant portion of the available hardware resources on the FPGA. The most used type of primitive blocks are the Lookup Tables (LUTs), which is quite expected, as all the logic is implemented using these small storage

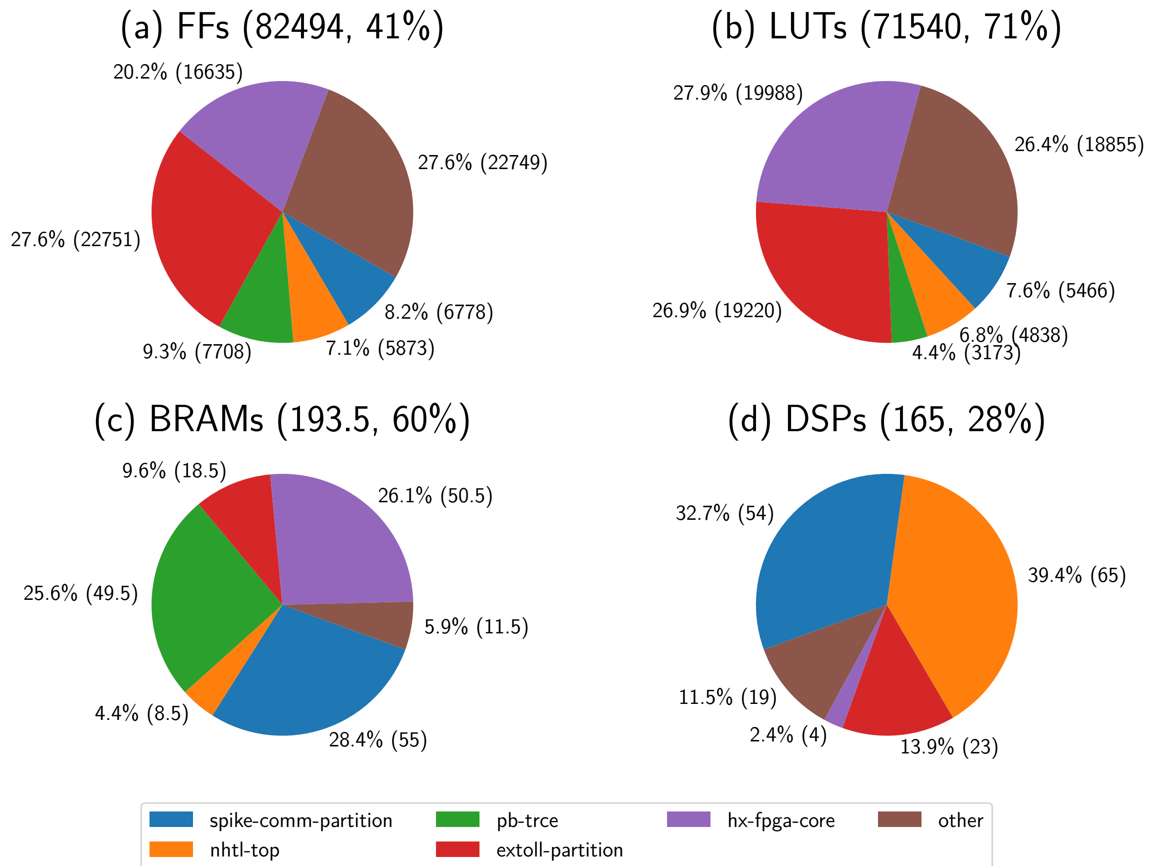


Figure 8.5: Absolute and relative primitive usage numbers of FFs (a), LUTs (b), BRAMs (c) and DSPs (d) for the overall FPGA design with parameters $S = 2$ and $B = 8$. The distribution of these primitive usage numbers is given in pie-charts for the major top-level units, as listed in the legend.

elements on an FPGA. Regarding the distribution of hardware resources among the major top level design units, the SPIKE_COMM partition uses only a small fraction of the total number of Flip Flops (8.2 %) the Lookup Tables (7.6 %), but a significant fraction of the Block-RAMs (28.4 %) and Digital Signal Processors (32.7 %). Regarding logic (LUTs) and registers (FFs) the EXTOLL partition and HXFPGA_CORE partition are the largest units. Regarding the BRAMs, the HXFPGA_-CORE partition and playback-trace memory are equally large as the SPIKE_COMM partition. Last but not least regarding the DSPs, the NHTL and SPIKE_COMM partition are by far the largest units, followed by the EXTOLL partition while the HXFPGA_CORE is negligible here.

As the focus of this thesis is on the event communication architecture described in Chapter 7, Figure 8.6 shows the distribution of primitive resource usage on the FPGA with respect to this partition unit.

By far, the largest fraction of FFs, LUTs and DSPs is spent on the bucket buffers. This is simply explained by the fact that this is the most complex unit in the event communication architecture, as it contains the whole encoding logic respectively including an instantiation of the UT Encoder. Besides that, the bucket unit is replicated multiple times ($B = 8$) in the design, as in the current implementation each bucket adds another possible network destination to the events generated by the neuromorphic chip. Even if a bucket buffer could handle more than one destination, as analysed

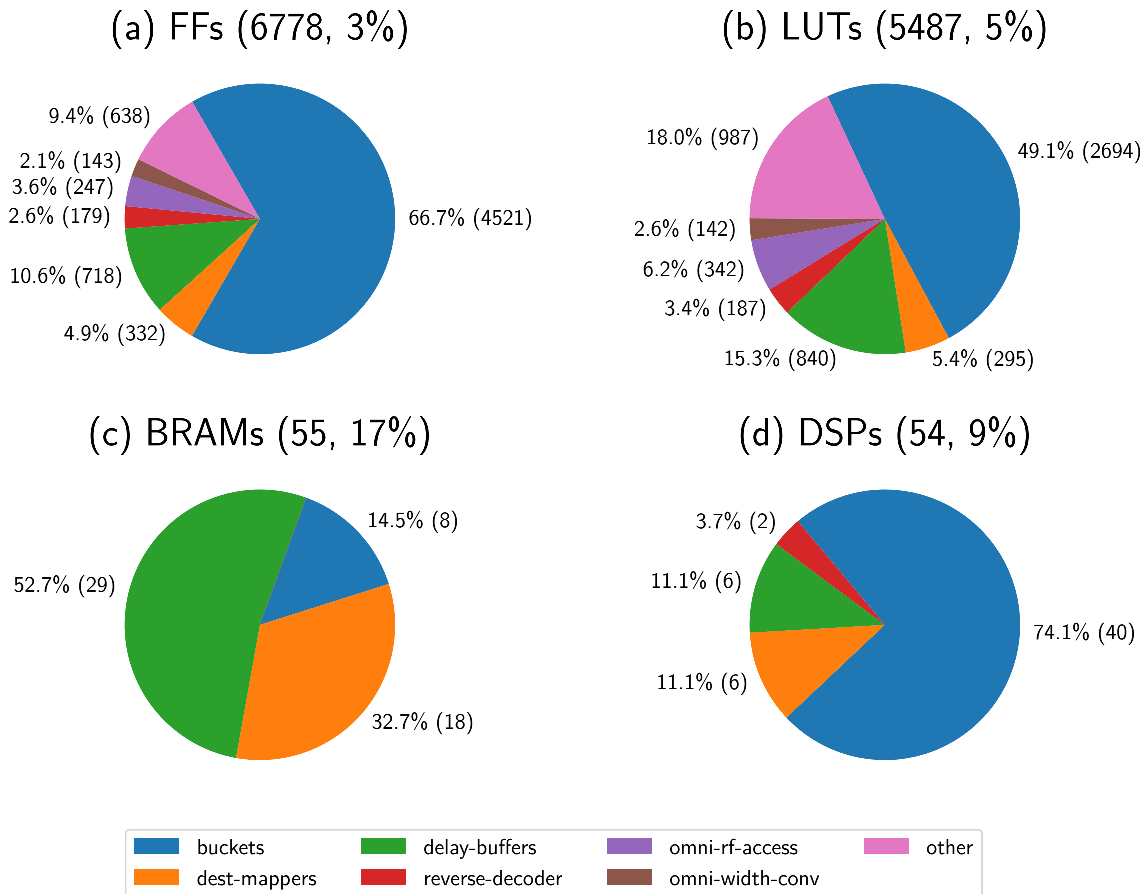


Figure 8.6: Absolute and relative primitive usage numbers of FFs (a), LUTs (b), BRAMs (c) and DSPs (d) for the SPIKE_COMM partition with parameters $S = 2$ and $B = 8$. The distribution of these primitive usage numbers is given in pie-charts for the major design units, as listed in the legend.

in Chapter 6, more bucket instances will imply less destination conflicts on the same bucket instance and thereby better performance, so the goal should always be to implement as much buckets as possible.

Regarding the Block-RAMs, the delay buffers use the most amount of resources, followed by the destination mappers and the buckets. The reason for this BRAM usage distribution is that the delay buffers implement the by far largest buffers, as they have to potentially store events for a long time (cf. Equation (5.10) and Section 7.5.3). The bucket buffers only implement a relatively small buffer of some few maximum sized packets. Especially, these buffers should be able to hold more than one packet in case the output is not immediately granted access due to other buckets currently requesting the output with temporarily higher RoundRobin priority. Still, these buffers are quite small as compared to the lookup tables (cf. Section 7.3.2) and delay buffers (cf. Section 7.5.3) with 2^{14} entries each.

The event data stream compressor, used to convert triple-event into double-events is reported by the Vivado® implementation tool to consume negligible (less than 1 %) amounts of LUTs, FFs and DSPs. This justifies the decision to favour this compressing unit above another parallel data path when compared to the requirement offset from $S = 2$ to $S = 3$ in Figure 8.4.

For an implemented number of 16 barrier- and 4 interrupt units, as provided in the EXTOLL Tourmalet network chip, the Vivado® tool reports approximately 0.8 % of the overall available FFs and approximately 1.3 % of the available LUT resources all together. Therefore it is not necessary to cut the number of units in order to save FPGA resources here and the full number of supported barrier- and interrupt units can be confidently implemented.

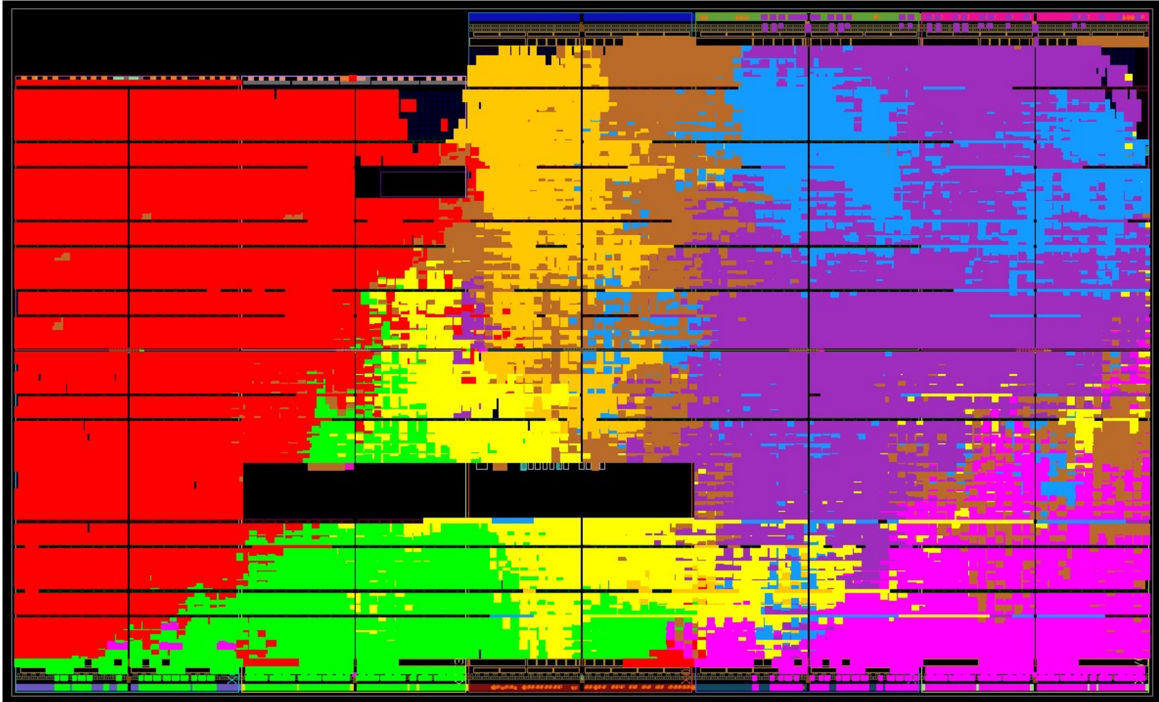


Figure 8.7: Screenshot of the implemented design floorplan in the Xilinx® Vivado® tool. The major design units are highlighted: **extoll-partition**, **nhtl-top**, **spike-comm-partition**, **hx-fpga-core**, **DDR3 memory interface for pb-trce**, **DDR3 memory interface for PPU extmem**, **AXI interconnect**, **others**. The colour coding is mainly the same as in Figure 8.5.

8.2.2 EXTOLL Operation Frequencies and Bandwidths

In order to harness the full provided bandwidth of the EXTOLL link between the BrainScaleS FPGA and the EXTOLL Tourmalet network card, the operation frequencies in the implemented FPGA design have to be larger than a certain minimum value. This value depends on different parameters like the width of the internal data paths and the operation frequency on the Tourmalet chip itself, as well as the physical configuration of the link, especially regarding the number of connected lanes (cf. Section 4.1.1). What counts in the end for a functional link connection, is that the lane rates R_l^{T} (Tourmalet) and R_l^{F} (FPGA) match on both ends of the physical connection:

$$R_l^{\mathrm{T}} = R_l^{\mathrm{F}} \tag{8.1}$$

Thereby this lane bandwidth R_l can be calculated from the network device's link frequency f_L and data path width w, as well as the number of connected lanes N_l

$$R_l = \frac{f_L \cdot w}{0.8 \cdot N_l} \tag{8.2}$$

where the factor 0.8 arises from the 8 bit-10 bit encoding that is used to secure the physical transaction words on each lane.
Equation (8.2) can now be applied to Equation (8.1) and solved for the required FPGA link frequency, leading to the following expression:

$$f_L^{\mathrm{F}} = f_L^{\mathrm{T}} \cdot \frac{w^{\mathrm{T}}}{w^{\mathrm{F}}} \cdot \frac{N_l^{\mathrm{F}}}{N_l^{\mathrm{T}}} \tag{8.3}$$

As the width of the core data path is 128 bit in both the Tourmalet and the BrainScaleS FPGA, the fraction $\frac{w^{\mathrm{T}}}{w^{\mathrm{F}}} = 1$. As stated in Section 4.1.1, the Tourmalet network chip offers $N_l^{\mathrm{T}} = 12$ lanes at each link. However, the BrainScaleS FPGA can only implement $N_l^{\mathrm{F}} = 4$ lanes due to limited availability of GTX transceivers for the EXTOLL connection. Thereby one arrives at the expression

$$f_L^{\mathrm{F}} = \frac{f_L^{\mathrm{T}}}{3} . \tag{8.4}$$

So in summary, as the FPGA can only make use of one quad on the link, only one third of the overall link data rate can be implemented. Therefore the FPGA also only needs an operation frequency at its EXTOLL link that equals a third of the Tourmalet's link frequency. As the Tourmalet boards, used in this project run at a frequency of $f_L^{\mathrm{T}} = 600\,\mathrm{MHz}$, this leads to a required FPGA frequency of $f_L^{\mathrm{F}} = 200\,\mathrm{MHz}$.
In order to ease the timing requirements on the FPGA design (cf. the description of general timing constraints in Section 7.7), the data rate at the Tourmalet link is reduced through configuration by a factor of two, leading to a remaining frequency requirement of

$$f_L^{\mathrm{F}} = 100\,\mathrm{MHz} . \tag{8.5}$$

Plugging these numbers back into Equation (8.2), the full EXTOLL link rate for communication between two Tourmalet cards derives to $R_L^{\mathrm{T}} = 96\,\frac{\mathrm{Gbit}}{\mathrm{s}}$. However, with the modifications for connecting to a BrainScaleS FPGA, the total link rate is reduced to a number of $R_L^{\mathrm{F}} = 16\,\frac{\mathrm{Gbit}}{\mathrm{s}}$. Again re-correcting this for the 8 bit-10 bit coding, the FPGA has a total useable EXTOLL information bandwidth of $R_{\mathrm{info}} = 12.8\,\frac{\mathrm{Gbit}}{\mathrm{s}}$. Taking into account the maximum protocol efficiency of 93.9 % (cf. Equation (B.3)), calculated from the RMA header overhead, one finally arrives at a useable bandwidth of

$$R_{\mathrm{rma}} = 12.0\,\frac{\mathrm{Gbit}}{\mathrm{s}} . \tag{8.6}$$

However, as pointed out in (J. Schmitt 2017), running the EXTOLL network partition such a low frequency as compared to the Tourmalet ASIC implementation, running at $f_E^{\mathrm{T}} = 600\,\mathrm{MHz}$, introduces a performance bottleneck in the network protocol flow control mechanism. This is caused by the restricted number of available credits which is now processed much slower in the FPGA implementation. Therefore, it is desirable to run the EXTOLL partition at a frequency higher than

the minimum value, derived from the lane rate equations above. The EXTOLL partition logic is therefore implemented at a frequency of

$$f_E^{\mathrm{F}} = 150\,\mathrm{MHz}\,. \tag{8.7}$$

As can be seen in Figure 3.4, the `HXFPGA_CORE`, as well as the `SPIKE_COMM` partition may provide a maximum amount of 64 bit of data per clock cycle at the BrainScaleS FPGA system frequency of 125 MHz. This corresponds to a data rate of $8\,\frac{\mathrm{Gbit}}{\mathrm{s}}$ respectively.

Consequently, each of these two data streams can be handled comfortably by the EXTOLL link throughput bandwidth of $12\,\frac{\mathrm{Gbit}}{\mathrm{s}}$. However, both streams cannot be sent simultaneously at full data-rate without bandwidth shortage. This conflict is in principle solved by sending the realtime spike communication stream with priority over the trace data stream from the *Executor*. However, the trace memory buffer in the FPGA will eventually run full, causing the *Event Switch* (cf. Section 7.1.5) to block both streams, in case the incoming event stream is replicated to both outputs.

8.2.3 The Test Setup

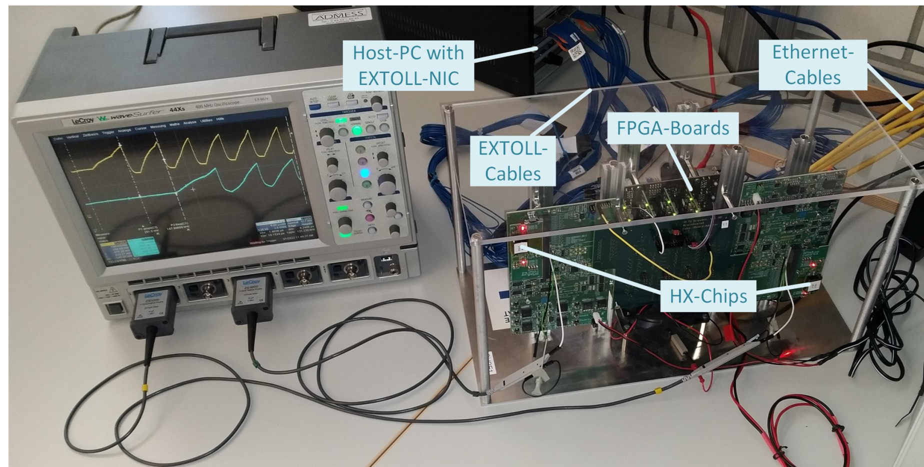


Figure 8.8: Photo of the prototype experiment setup using the EXTOLL network topology for spike event- and host communication.

For the tests and measurements, described in this Chapter, the current BSS-2 test platform was used, which has been described in Chapter 3. A photograph of the particular test setup, used for this thesis is shown in Figure 8.8. The adaptations made for connecting it to the EXTOLL network are the same as in (Thommes 2018). The EXTOLL network is connected to the FPGA communication boards via special EXTOLL cables, manually equipped with USB 3.0 plugs, one per lane. The pin-layout of these plugs has been documented in (Thommes 2018). Additionally, for the purpose of programming bitstream files to the FPGA boards, the setup is still connected to an Ethernet network. For the purpose of measuring inter-chip spike latencies (cf. Section 8.5) and the synchrony precision of the EXTOLL interrupt (cf. Section 8.6.3), an oscilloscope is attached to GPIO pins on the setup.

8.3 Network Operation Tools

In order to bring up the EXTOLL network and to do basic accesses to remote Registerfiles from a host computer's perspective, some basic, but essential tools and configurations are required. These include the configuration of the FPGA-facing links on the Tourmalet ASICs in the network, discovery of the physically connected topology, configuration of the routing network tables, utility executables for accessing remote Registerfiles and the refreshing of lost identity information on powercycled FPGAs.

These tools and tasks have partly been implemented and solved in the scope of this thesis and will be shortly introduced and described in the following subsections.

8.3.1 Link Configuration

When powercycling the Tourmalet network card, the EXTOLL ASIC will automatically attempt to establish a connection across all links, where an attached cable is detected. This connection is thereby established, by going through a link training procedure which finds the correct delay settings in a Delay-Locked-Loop (DLL) (cf. e.g. Xanthopoulos 2009) by transmitting known test patters across the link. This DLL ensures that both communication partners are synchronised across the link and can exchange code words in a synchronous way. However, when the communication partners at both ends have different physical parametrisations of their link, the link training procedure will not find a working delay setting. These fundamental link parameters include for example different numbers of lanes and data rates or different data path widths as it is the case with the EXTOLL Tourmalet and the BrainScaleS FPGA (cf. Section 8.2.2). When initialising the network, the Tourmalet links that are physically connected to an FPGA node therefore have to be configured correctly in order to setup a working connection between both nodes. This configuration includes the deactivation of two out of three lane-quads as well as the throttling of the link speed by a factor of two, as motivated in Section 8.2.2. After the Tourmalet link has been configured correctly to match the FPGA's parametrisation, the link training procedure has to be retriggered and will now be able to find the a working delay setting to establish a communication connection across the link.

This link configuration has to be done for each Tourmalet node in the network that connects to BrainScaleS FPGAs. The configuration is thereby written to the local or remote Registerfile of the respective EXTOLL ASIC either by accessing the Configuration space through the driver-supported memory map, or through remote Registerfile access messages (cf. Section 8.3.3). The latter is possible, as the host computer's driver installation will execute an initial run of the EMP tool (cf. Section 8.3.2) at system startup, already setting up the network operation with the subset of nodes that are default-compatible to each other.

8.3.2 The Extoll Management Program

When all links have established a functioning communication connection, the next step is to configure the routing tables in the network crossbars in every switching network node (cf. Section 4.1.3). This task is executed by the Extoll Management Program (EMP) tool which is provided by EXTOLL whose function has been summarised in Section 4.4.

Especially its first and third step of operation, i.e. discovering the topology and writing the routing tables using temporary routing configurations, are quite traffic intensive and are not robust against unstable link connections. So if a link connection is not stable and (temporarily) breaks during this procedure, the EMP run will fail.
During this thesis work, it turned out that the links between a Tourmalet ASIC and the BrainScaleS FPGAs can be quite unstable after the Tourmalet has been powercycled and the FPGAs have been not. The links only become stable when adhering to a strict powercycle procedure, where the FPGAs are powercycled together with the Tourmalet ASIC (via powercycle of the host-computer) *before* the EMP tool is run.
As BrainScaleS FPGAs are always leaf nodes to the network, i.e. don't possess a crossbar switch and therefore also don't possess a routing table, prone to loss of state during powercycle, the EMP tool is not needed to be re-run after FPGAs have been powercycled.

8.3.3 EXTOLL Registerfile Access Utilities

There are two paths to access a configuration and status Registerfile at a local Tourmalet node. Hereby *local* is defined relative to the host computer on which the access is to be performed. A network card is called *local* to a host computer if it is connected via the PCIe bus and enumerated in the kernel driver. It has to be noted that multiple network cards can be physically connected to the PCIe bus, but only one of them may be handled by the driver. Every additional network card that is physically connected to the host's PCIe bus is only supplied with power through the PCIe connector and will exclusively be reachable via an active network connection to the driver-supported primary network card.
A local Registerfile will be automatically mapped by the EXTOLL driver into the file system of the host computer. This means that every register address is mirrored by a respective file and may be accessed through OS-driven file accesses. An exception to this rule are RAM blocks in the registerfile, which are mapped to a single file respectively. Additionally for each register, there are two files; one containing the raw hexadecimal value, and one printed in a human readable format containing the values of the individual fields. In case of RAM blocks, the files contain a single line for each RAM address. The paths to those files resemble the hierarchy of the Registerfile, as defined by the TCL source and distributed throughout the hardware hierarchy (cf. Section 4.2.4 and Section 3.2.5.2).
The second way to access a Registerfile is via RMA messages, marked by an active RRA modifier bit (cf. Appendix B.2). These are initiated through a software descriptor, which is communicated via the PCIe bus to the RMA hardware unit (cf. Section 4.2.1). The RMA unit then initiates a network packet targeting the remote RMA unit, which in turn accesses the respective registerfile address and responds to the sending node's RMA unit. Notably, this not only works for remote nodes, but also locally, as RMA messages that are addressed to the local node will not be sent across the network. Instead, the local RMA unit will directly handle the local Registerfile access, as if it had just arrived from another node on the network. Thereby, the whole network, including the local node, can be configured solely via RRA messages.
In order to make these two Registerfile access paths easily available, a small Python utility library has been written. For both access paths, the utility library supports reading and writing to specific

registerfile addresses or hierarchical path specifiers respectively. Additionally, there are convenience functions to get or modify specific bit fields at the retrieved register content. In case of filesystem mapped accesses, the read and write functions operate on the raw valued files. In case of RAM blocks, the access functions require an additional input argument regarding the requested line.

The RMA based remote Registerfile access utilities rely on a two small binary executables (`rra_read` and `rra_write`), available trough the `lib_rma` software library, which is provided by EXTOLL (Electronic Visions(s), Heidelberg University n.d.[i]). These are run as subprocess from the Python code which then parses the retrieved console output in case of read access. Notably, if the remote node is not reachable, either because it is powered down or it's not reachable through the network due to a broken link connection, the `rra_read` executable will not return and therefore has to be aborted either by user interaction using `Ctrl-C` or by an expired timeout of the subprocess. In the latter case, the Python utility will throw an appropriate exception up the call-stack.

8.3.4 Refreshing of IDs

When powercycling BrainScaleS FPGAs, e.g. after flashing a new bitstream the implemented design looses any state that was imprinted from outside before the power reset happened. This especially includes the own NDID that was initially assigned by the EMP tool, run at a master network node. However, some mechanisms in the FPGA design, as e.g. the Omnibus to Registerfile access bridge (cf. Section 7.6.4.2), as well as the RMA network protocol, implemented by the NHTL unit (cf. Section 7.4 and (Thommes 2018)), rely on this information about the local NDID. In order to restore this functionality of the design, the register containing the NDID has to be re-written on all powercycled nodes. In principle, this could be done by simply re-running the EMP tool. However, it turns out that this is not quite stable and will often result in a non-functional network, as the EMP run will re-discover and re-write the whole network configuration which might trigger some hidden bug. Instead, as the topology did not change through the FPGA's powercycle and the driver and Tourmalet ASIC still have a correct representation of the network, one can simply write the respective register on the FPGA with the NDID information that can be retrieved from the EMP tool (cf. Section 8.3.2).

It has been observed throughout this work that after rebooting an FPGA, an RRA read operation will not succeed. However, when investigating this problem using the ChipScope method, described in Section 8.1.5.2, it can be seen, hat the NHTL correctly receives and responds to the read request message. When looking into the interface signals of the NP and LP on the FPGA, it is observed that the response message gets stuck in the NP. Further investigation inside the NP revealed that the FPGA's flow-control mechanism has run out of credits, implying that the credit return mechanism from the Tourmalet has somehow been broken by the powercycle of its partner FPGA. It was further observed that the FPGA will again be reachable after another powercycle, but only if an access was attempted in between.

However, this non-reachability can completely be avoided if the FPGA's LP is requested to send its ids to the partnering Tourmalet node before the powercycle while it is still reachable. In order to prevent the FPGAs to not being reachable after powercycle, this workaround has been included in a utility script together with the refreshment of the NDID, which is run after every powercycle.

8.4 Software Integration

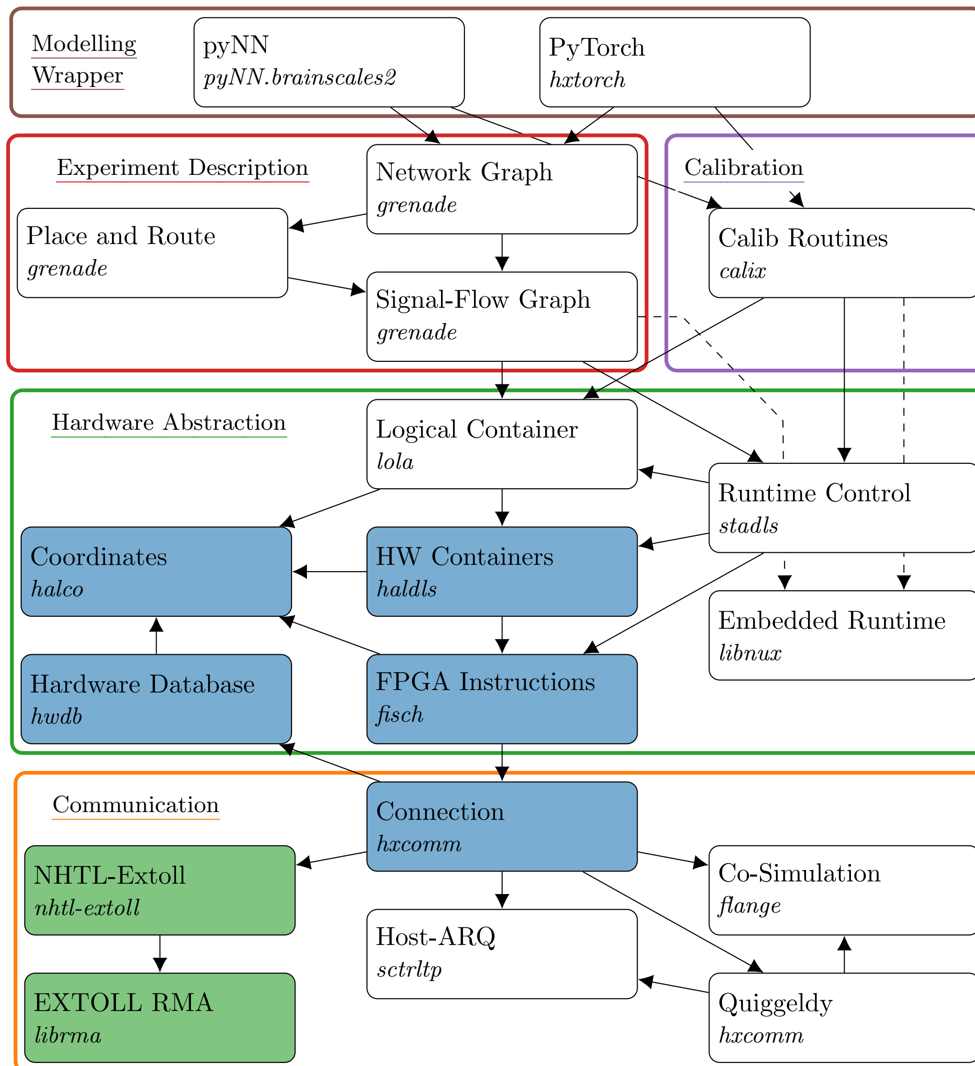


Figure 8.9: Overview of the BSS-2 software architecture. Libraries with white background are unchanged from the original software stack (cf. Figure 3.5), blue libraries have been extended and green libraries were added during this work.
This Figure has been modified from (Müller, Arnold, et al. 2022).

In order to integrate the communication functionality added to the BrainScaleS FPGA's hardware design (cf. Chapter 7) into the BSS-2 software stack (cf. Section 3.3), two communication layers have been added. These are indicated with green background at the bottom part of Figure 8.9. The *libRMA* (Electronic Visions(s), Heidelberg University n.d.[i]) is provided by EXTOLL and implements the basic API for using EXTOLL's RMA communication paradigm and protocol, supported by the RMA unit on the Tourmalet ASIC (cf. Section 4.2.1) and the NHTL unit on the BrainScaleS FPGAs respectively (cf. 7.4). The *NHTL-Extoll* layer builds upon the *libRMA* and implements the communication protocol of the NHTL unit, defined in (Thommes 2018).

In addition to these new layers, existing software layers had to be adapted in order to integrate the EXTOLL communication and additional configuration and status readout capabilities into the software stack. These layers are indicated with blue background in Figure 8.9.

With respect to the EXTOLL software stack, which has been summarised in Section 4.4, the *NHTL-Extoll* and *hxcomm* libraries act as middleware and all higher layers above them represent a user application.
The following Sections go into detail about the important aspects of these added and adapted software layers.

8.4.1 The Neuromorphic Hardware Transaction Layer via Extoll

In order to have the BrainScaleS software stack make use of the EXTOLL network, a new layer is needed that implements the specific protocol details, defined by the NHTL FPGA hardware unit (cf. Section 7.4), while using the API provided by the EXTOLL software stack and again providing a simple API for the *hxcomm* library of the BrainScaleS software stack. While the BrainScaleS software stack has been summarised in Section 3.3, the EXTOLL software stack is summarised in Section 4.4.
A bridge between the EXTOLL RMA library (*libRMA*) and the BrainScaleS application is provided by the *Neuromorphic Hardware Transaction Layer via Extoll (NHTL-Extoll)* library. Originally an according software layer has been developed for BrainScaleS-1 by (N. A. Buwen 2019), called *libHBP* (cf. Thommes, N. Buwen, et al. 2021). With the help of this existing state, Sven Bordukat developed the NHTL-Extoll library and integrated it into the BSS-2 software stack (cf. Thommes, Bordukat, et al. 2022). In the following, this library will be referred to as *NHTL-Extoll*. However, the respective operation principles and algorithms also apply to the *libHBP* from (N. A. Buwen 2019) accordingly.
As already mentioned, the main purpose of this library is to implement the host-side part of the NHTL transport layer protocol, which is summarised in Figure 8.10.
The *libRMA* operates on virtual *connections* attaching to an opened *port*. The *port* is associated with a Virtual Process ID (VPID) while the *connection* is associated with the respective remote Node ID (NDID) and connection parameters like whether the connection performs Remote Registerfile Accesss or whether it operates on physical or virtual addresses. The *NHTL-Extoll* library defines the entity of an *Endpoint* referring to a specific remote BrainScaleS FPGA, connected via the EXTOLL network. The *Endpoint* thereby holds two virtual connections to the NDID of the respective FPGA. One connection for RRA and another for RMA data transfers.
As the remote FPGA directly transfers its data into main memory regions of the host, the *Endpoint* holds management objects for memory regions that are registered at the RMA driver via the according *libRMA* API function. According to the NHTL protocol (cf. Section 7.4 and Figure 7.6), any data that is not an RRA response is written to a special memory region that is operated in a ringbuffer fashion and therefore has to be registered to the FPGA regarding its start address and size. The *NHTL-Extoll* library ensures the correct initialisation of this ringbuffer memory region.
As the NHTL hardware unit will notify the availability of new data in that ringbuffer to the host using special notification messages (cf. Figure B.6), the *NHTL-Extoll* software library has to implement some mechanism to receive and react to these notifications. For this purpose, the *libRMA* offers API functions queries to the EXTOLL notification buffer for newly arrived notifications. The *NHTL-Extoll Endpoint* will start an independent active polling thread that calls the respective query function in regular intervals and goes to sleep in between. If a notification, delivering information

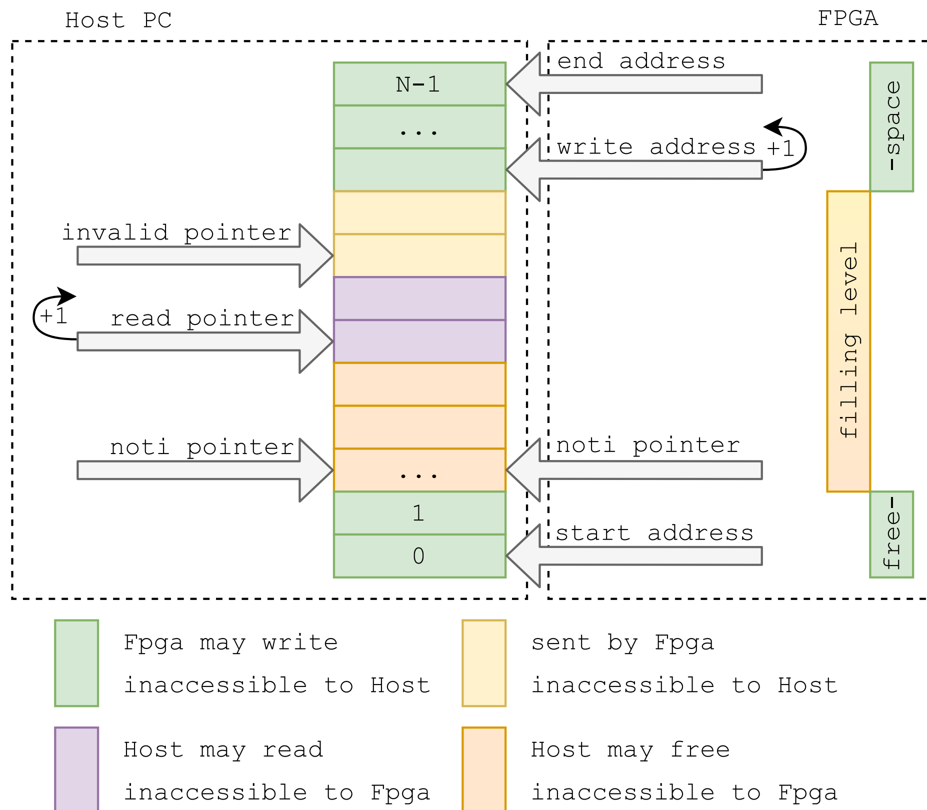


Figure 8.10: Visualisation of the NHTL transport layer protocol between the host and an FPGA. The Figure has been taken and modified from (N. A. Buwen 2019). A hardware unit on the FPGA holds pointers to the `start`- and `end addresses` of a host-based memory region as well as a `write pointer` and a representation of the amount of free space (`noti pointer`). The host on its side holds a `read pointer` as well as a representation of the amount of readable, i.e. valid content (`invalid pointer` and `noti pointer`). From time to time, the FPGA notifies the host about new valid data it has written to the memory and the host in turn notifies the FPGA about the amount of space that can be overwritten as it has recently been read. For details, cf. Section 7.4 as well as (Thommes 2018) and (N. A. Buwen 2019).

about available data in the ringbuffer is received, the respective management object is updated and the received data is made available to the API towards the higher software layer.

8.4.2 Round Trip Latency of the NHTL protocol

Using the AL-Test Interface, described in Section 8.1.5.4, the round-trip latency of the NHTL-Extoll transport layer has been measured. For this purpose, the AL-Test Interface was configured for loop-back operation. The host then sends small messages to the FPGA where they are mirrored back to the receiving ringbuffer in the host's memory. After every message transfer, the host waits for the notification message from the NHTL unit that the mirrored response should have arrived at the ring-buffer. This round-trip measurement was then repeated for 10^7 and 10^8 times in two experiments respectively.

From the resulting histogram plot in Figure 8.11 one can see that the minimum latency is in a bin range between 4.5 µs and 10 µs, the median latency is located around 50 µs and the typical maximum

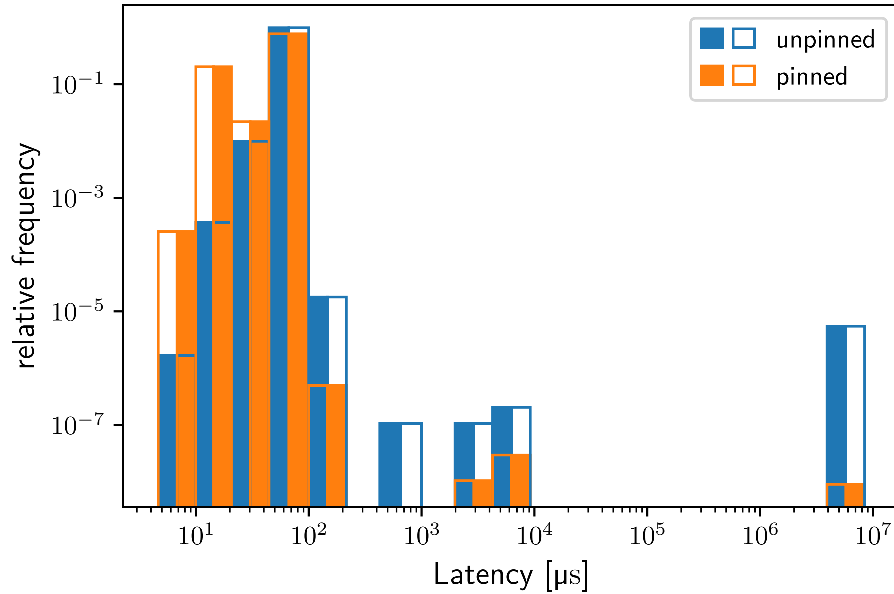


Figure 8.11: Result histograms to the latency measurement of the transport layer protocol. The histogram shows the relative frequencies of the respective measured latency values. The results have been obtained with pinned and unpinned software threads respectively.

latency (where most measurements lie below) is between 100 µs and 200 µs. However there are some outliers of up to 10 ms, and most notably a significant amount of extremely high latency runs of about 5 s to 10 s has been observed at a relative occurrence rate of 10^{-5} in the *unpinned* case.

As the NHTL unit triggers the notification message using a timeout counter which is reset whenever a data transaction is acquired for sending towards the host and only a single Quad Word transaction is sent across the loopback, the minimum latency is limited by the value of this timeout, which was configured to 2 µs. The actual minimum for the half-roundtrip transport latency can therefore be calculated by subtracting this timeout value from the minimum measured round-trip latency value:

$$\min\left(l_{\text{hrt, transport}}\right) = \frac{\min\left(l_{\text{rt, measured}}\right) - l_{\text{timeout}}}{2} \tag{8.8}$$

Thereby one arrives at a value of around $\min\left(l_{\text{hrt, transport}}\right) = 1.25\,\mu\text{s}$ to $4\,\mu\text{s}$ for the minimum half round-trip latency. This nicely matches the numbers given in the EXTOLL literature of (Nüssle, Scherer, et al. 2009), stating a half-roundtrip latency of down to 1.2 µs for 8 B of payload.

While the general distribution of latencies seen in Figure 8.11 is justifiable with the varying overhead of the operating system and CPU scheduling, the enormous latency peaks of several full seconds need to be further investigated.

An unstable link connection between the host and the FPGA has been excluded by looking at the respective error counter registers (cf. Section 8.1.5.3) in the Registerfile of the Tourmalet ASIC's and FPGA's Link-Ports. By reading the performance counter registers for received notification messages in the Tourmalet ASIC's RMA unit after waiting for some expectable latency time, it was confirmed that said messages actually arrive within "normal" latency periods. Also the actual data, which is transmitted in a separate message before the notification, is present in the ringbuffer memory.

These findings imply that the observed enormous latencies do not arise from the hardware imple-

mentation of the NHTL- or the RMA protocol, but rather from somewhere in between the Tourmalet ASIC's RMA unit and the user space software. A significant improvement could be achieved by pinning the user space software threads (the main test execution thread and the thread polling for new notification messages in the queue) to the same CPU core. Thereby, the relative occurrence rate could be decreased to 10^{-8} which can be seen from the *pinned* histogram in Figure 8.11.
From the comparison of both the unpinned and pinned histograms, one can infer that with the CPU pinning in place, not only the relative frequency for the extreme outliers decreased significantly by a factor of 10^3, but also the "normal" outliers at around 1 ms decreased by an approximate factor of 10. In return, the relative frequency for minimal and small latencies (the first three bars in the histogram) significantly increased. From this observation it can be reasoned that the cause of high latencies lies in the thread-level communication and synchronisation between different CPU cores, which could e.g. be implemented by cache coherency protocols or atomic operations in the CPU. As the kernel driver has not been pinned to the same CPU core, the remaining outliers could be attributed to the synchronisation across this process border. However it is not understood, why this might possibly take multiple seconds.
The tests have been performed on an *Intel® Xeon® Silver 4108* CPU.

8.4.3 The *hxcomm* Communication Layer

The first layer in the previously existing BrainScaleS software stack that has to be adapted to the new NHTL-Extoll transport layer is the *hxcomm* library. This library defines a bunch of specific connection classes, implementing a common interface to the different transport layers. In the existing stack these are two alternative transport layers, namely the *Host-ARQ*, which is based on UDP via Ethernet, and the *flange* Co-Simulation interface layer. An *hxcomm* connection object thereby serves as a basis for any higher level communication with the BrainScaleS system. The different connection classes are implemented in a way, such that the specific type of the connection is interchangeable in higher software layers.
For the new NHTL-Extoll transport layer, a new connection type has been implemented which uses the API provided by the NHTL-Extoll library. The connection object thereby holds an *Endpoint* object for a specific remote FPGA node id, as described in Section 8.4.1. If no node-id is explicitly given to the connection constructor, it retrieves one from a list provided by the NHTL-Extoll library. This list is in turn retrieved from a call to the Extoll Management Program. The *hxcomm* library also offers convenience functions, returning a connection to an FPGA identified by variables defined at the OS environment. As these environment variables contain a generic system specifier, this has to be converted into a node-id (or IP-address in case of a Host-ARQ) before constructing the connection object. This conversion is supported by the information stored in the *Hardware Database*.
The *hxcomm* library also offers the possibility to automatically return the respectively available transport layer by means of checking the OS environment.

8.4.4 The *halco* Coordinate Layer

On the next upper level in the BrainScaleS software architecture, abstraction of the hardware is implemented in several layers.

Generally, the hardware abstraction in the BrainScaleS software stack makes use of the concept of coordinates and containers which is explained in detail in (Müller, Mauch, et al. 2020). Coordinates define an abstract virtual address space on the system's physical configuration and status address space using custom defined ranged integer types. Containers on the other hand are objects that model the internal structure and actual addresses of the configuration and status register accesses. For the calculation of specific addresses, the container classes rely on the information passed through a specific coordinate value of the correct type. Container objects thereby represent "a possible state of a specific hardware entity or entity group". (Müller, Mauch, et al. 2020)

The general BrainScaleS software API defines a playback program for the FPGA's executor unit as a sequence of write- or read instructions at specific coordinates, consuming or producing container objects, accompanied by control flow instructions.

The *halco* software layer defines and implements the coordinate types and ranges that are used by container classes from both *fisch* (Section 8.4.5) and *haldls* (Section 8.4.6), as well as their hierarchical dependencies relative to each other.

In order for being able to access EXTOLL Registerfile locations at any node in the network, using the Omnibus-Registerfile access bridge (cf. Section 7.6.4), a low level coordinate structure has to be defined that includes both Registerfile addresses and node-ids. These low level coordinates will then be used by the *fisch* library (cf. Section 8.4.5).

For this purpose, a coordinate type for 64 bit wide addresses is defined (`ExtollAddress`), as well as one for 16 bit wide node-ids (`ExtollNodeId`). However, some Registerfile addresses are only present on Tourmalet ASICs and others are only present on BrainScaleS FPGAs. Furthermore, there are also locations that, although they are logically present on both node types, are located at different addresses on either Tourmalet ASIC or FPGA. To account for this fact, an additional coordinate type is needed for the node type, so far only implementing two enumerated values for the Tourmalet ASIC and the BrainScaleS FPGA (`ExtollChipType`). Having defined these basic coordinate types, further compound coordinate types are derived. One type combines the `ExtollNodeId` with the `ExtollChipType` to the new type `ExtollNodeIdOnExtollNetwork`. Another type further combines the `ExtollAddress` with this compound type to a new compound coordinate type named `ExtollAddressOnExtollNetwork`.

High level coordinates are defined for each configuration location on the system. These high level coordinates will then be used by the *haldls* abstraction layer (cf. Section 8.4.6). If they are located in an EXTOLL Registerfile somewhere on the network, the high level coordinates are concatenated with the `ExtollNodeIdOnExtollNetwork` coordinate type. Generally, all compound types have member functions providing conversions to one of the original types.

For the spike event communication architecture, as well as for any existing design hierarchy in the BrainScaleS system, the same method of concatenating coordinate types is used to create a hierarchical coordinate structure. For example, a basic coordinate type `ExtollSpikeCommSplitOnFPGA` is defined, modelling the two parallel data paths in the spike communication architecture (cf. Section 7.8).

8.4.5 The *fisch* Instruction Layer

The *FPGA Instruction Set Compiler for HICANN (fisch)* abstracts arbitrary FPGA instructions, implemented by the communication layer (*hxcomm*), to a form of register-like read and write instructions. This is done in the form of container classes. For example, this includes containers for creating Omnibus or JTAG transactions, timing instructions or the creation of spike event stimuli. Each container class is directly referenced to a corresponding coordinate type.

In the context of this work, *fisch* containers have been added for access transactions to the EXTOLL Registerfile via the Omnibus (cf. Section 3.2.5 and Section 7.6), as well as for the new barrier- and interrupt instructions (cf. Section 7.2.2). The former defines a Registerfile access with 64 bit payload and performs two Omnibus accesses to two subsequent Omnibus addresses according to the requested Registerfile address. The data-width converter unit on the FPGA will collect these Omnibus transfers and convert them into a single Registerfile transfer (cf. Section 7.6.2). There are two variants of this container class. One is solely for Registerfile accesses on the local FPGA, while the other also supports remote Registerfile accesses, mastered by the local FPGA playback executor. In order to calculate the respective address, the respective member function of the container takes a coordinate instance of the respective type. For example, for local-only accesses, this is an `ExtollAddress` coordinate, while for network-global accesses an `ExtollAddressOnExtollNetwork` is required.

For the new barrier- and interrupt wait-instructions, the existing wait-until instruction was amended by two new variants accordingly.

8.4.6 The *haldls* Abstraction Layer

This hardware abstraction layer defines and implements container classes modelling the detailed structure of the configuration and status space on the BrainScaleS system. This exemplarily includes configuration containers for the neuron and synapse circuits on the BrainScaleS ASICs as well as for settings on the FPGA. These high level containers are linked to one or multiple of the basic access containers in the *fisch* (Section 8.4.5) library. Every *haldls* container offers a member function that calculates the respective low-level address coordinate, using an input of its associated high-level coordinate type.

In the scope of this work, new containers have been added in the *haldls* layer for Registerfile locations related to configuration and status readout of the barrier- and interrupt units (cf. Section 4.1.2 and Section 7.2.2) as well as the spike event communication architecture, described in Section 7.3 and Section 7.5. Also some native Omnibus configuration containers were added for the event switching unit, described in Section 7.1. These containers thereby also rely on newly added coordinate types (cf. Section 8.4.4).

The *haldls* library also implements an extensive test library that tests every software container and the underlying hardware implementation for consistency with respect to data integrity after writing and reading back configurations. This test-suite has been successfully executed via the NHTL-Extoll transport layer, thereby also verifying the correct implementation of the NHTL hardware transport layer on the FPGA.

8.4.7 The *calix* Calibration Layer as NHTL Stress Test

On top of the hardware abstraction level, the *calix* library implements algorithms and routines for calibration of the BSS-2 neuromorphic hardware system, mainly concerning the analogue circuits on the HICANN-X ASIC. Although this layer has not been changed in the scope of this work, it yet serves as an extensive stress test to the NHTL-Extoll and NHTL transport layer, as the calibration routines perform lots of playback program execution cycles and transfer loads of measurement and calibration data for the extensive analogue configuration space of the BSS-2 ASIC.
Despite that the execution of the main calibration routine has initially re-exposed and suffered from the extensive latency spikes observed and described in Section 8.4.2, finally the calibration routine has successfully and reliably been executed via the EXTOLL transport layer.
Thereby, probably a notification message being unnoticed by the libRMA could not trigger the release of space in the host's receiving ring-buffer. This would eventually lead to stalling the data transfer from the FPGA's *playback executor* to the host, also stalling the execution of the playback program, as more data arrives from the HICANN-X ASIC. This finally lead to expiration of the configured timeout value (defaulting to 0.1 s) in the *playback executor*. Accordingly, the problem could successfully be hidden by simply increasing this timeout value to around 5 s in the calibration routine. However, this is not a viable work-around as it would only hide the symptoms of the actual problem, while potentially also hiding unrelated problems that would not be detectable with such an excessive timeout.
Instead of increasing the timeout value, the underlying problem could be further significantly rarefied by strongly decreasing the notification rate from the NHTL. Here the underlying mitigation mechanism builds on the fact that for the test in Section 8.4.2 the obtained occurrence rates are absolute, as each message round-trip will wait for the respective notification. However, when the rate of notifications per number of messages is decreased, also the occurrence rate of the observed problem per number of messages will decrease likewise.
In summary, with decreased notification rate, the calibration could successfully be executed without modifications.

8.5 Inter-Chip Latency Measurement

In order to obtain first results for the spike event latency between two chips, connected across an EXTOLL network, the postsynaptic potential (PSP) trace of a single neuron has been connected to an analogue readout pin of the BSS-2 ASIC and from there to a GPIO pin of the chip-carrier board (cf. Section 3.1.1 on page 35), from where they can be recorded using an oscilloscope. This has been done on two individual BSS-2 ASICs in order to compare the time-course of the respective PSP traces on an oscilloscope. The experiment thereby stimulates a neuron on the first chip which emits spike events that are sent across the network to a neuron on the second chip.
A screenshot from the oscilloscope during this measurement can be seen in Figure 8.12. The latency is now measured between the time where the PSP of the first neuron drops and the time where the PSP of the second neuron starts rising. In this example a delay of 1.90 µs is measured between the oscilloscope cursors. For these measurements, the accumulation timeout has been set to a minimum value and received events are immediately forwarded to the second chip without waiting for a

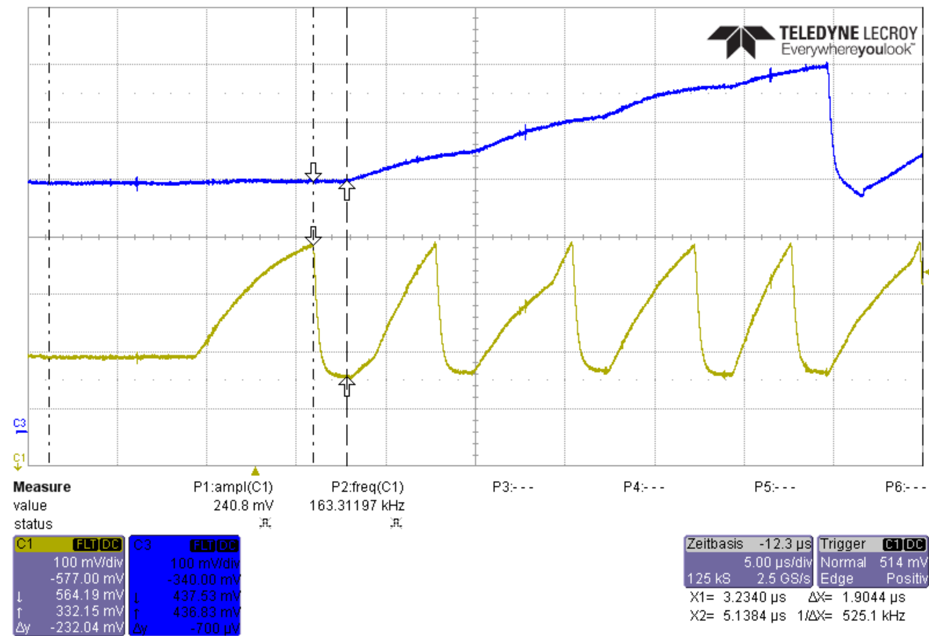


Figure 8.12: Screenshot on the oscilloscope measurement of the inter-chip spike event latency. The presynaptic spike-times are measured at the point where the voltage trace starts dropping while the postsynaptic spike-times are measured at the point where the respective voltage trace starts rising.

configurable axonal delay.

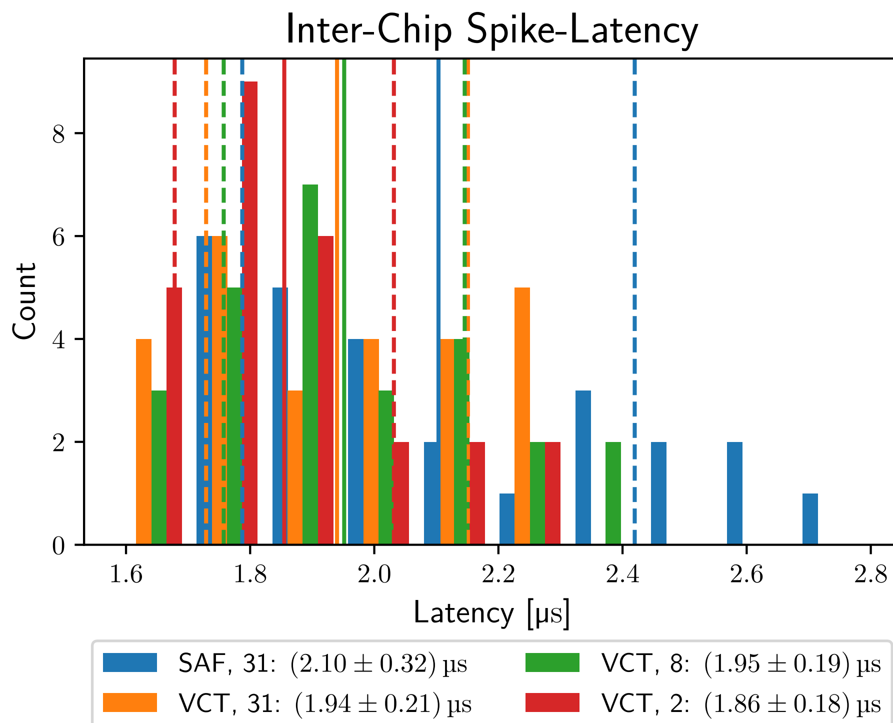


Figure 8.13: Histogram of the measured inter-chip latency with different settings of the EXTOLL NP on the FPGAs. These settings encompass whether packets are inserted to the network in SAF- or VCT mode and how many words are required in the NP's buffer in order to start transmitting the packet.

The described measurement is repeated multiple times with different settings of the EXTOLL NP on the FPGAs. These settings encompass whether packets are inserted to the network in SAF- or VCT mode and how many words are required in the NP's buffer in order to start transmitting the packet. As these settings directly control a packet's latency, they pose potential for optimisation with respect to the default values (SAF with 31 words required). These default values are of course not ideal in terms of latency, but were chosen in order to compact the transmission of packets to an end-to-end fashion. However, as the NHTL unit and the accumulation buffers already take care of this, the latency can be optimised here.

Results are plotted in histogram form in Figure 8.13. One can observe that the mean inter-chip latency improves by around 16 ns only through switching from SAF mode to VCT. Changing the amount of required words from 31 to 8 does not significantly change the latency, probably because the transmitted packets are smaller than this threshold and therefore a timeout takes precedence anyway. However, further lowering the threshold to 2 again improves the mean latency by about 9 ns. In total, the best optimised setting (*VCT, 2*) exhibits latency values in a range of

$$l_{\mathrm{ev}} = 1.61\,\mu\mathrm{s} \text{ to } 2.25\,\mu\mathrm{s} \quad . \tag{8.9}$$

It should be noted that generally the latency exhibits a standard variance between 18 ns and 32 ns, which also decreases in the direction of optimised settings. An explanation for these rather large latency variations is the interleaved transmission of spike event packets together with host-traffic containing e.g. MADC data traces, which have been enabled for comparison with the oscilloscope readings. If a spike event packet requests to be sent while a host-packet is already underway, the spike packet has to wait until the host-packet has been fully injected to the network, causing extra latency. As this extra latency also improves with the configuration optimisation, this also explains the observed trend in latency variation.

The expected numbers for the inter-chip transmission latencies have already been estimated in Section 7.5.3.1. The total inter-chip link-latency can be estimated between 512 ns and 868 ns according to (Karasenko 2020). This variation of course also contributes to the observed latency variation. When subtracting these reported values from the measured latency range, a remaining portion of the latency is obtained that can be attributed to the network transmission

$$l_{\mathrm{trans}} = l_{\mathrm{ev}} - l_{\mathrm{link}} = 1.098\,\mu\mathrm{s} \text{ to } 1.382\,\mu\mathrm{s} \quad . \tag{8.10}$$

The middle value of this range is 1.240 µs which is about $\frac{211\,\mathrm{ns}}{8\,\frac{\mathrm{ns}}{\mathrm{clk}}} \approx 26\,\mathrm{clk}$ higher than the expected network transmission latency of 1.029 µs according to Section 7.5.3.1. This deviation can be attributed to the additional latency caused by interleaved host traffic (0 clk to 64 clk due to the network MTU of 64 QW), which on average shifts the mean latency to higher values. To a small portion it will also be attributed to the additional latency of the spike event data path described in Chapter 7.

In summary, the measured latency range agrees well with the estimated network transmission latency in Section 7.5.3.1.

The results of these measurements have been presented in a spotlight presentation at the 9th Annual Neuro-Inspired Computational Elements (NICE) online workshop (cf. Thommes, Bordukat, et al. 2022).

8.6 Systime and Experiment Synchronisation

The concept of Barrier- and Interrupt operations, as developed by (Burkhardt 2007) and implemented in the EXTOLL hardware (Burkhardt 2012) has been summarised and explained in Section 4.1.2 of this thesis. Building on this concept, Section 7.2 introduced a method of synchronising a global systime across multiple FPGAs in an EXTOLL network using the provided global Interrupt functionality. As the EXTOLL Interrupt relies on precise measurements of the particular link latencies in the network, this Section will go into detail about these measurements. As the measurement routine for the Interrupt operation is not provided by EXTOLL at the current time, it had to be implemented in the course of this work. To our best knowledge, this is also the first characterisation of the EXTOLL global Interrupt operation. In (Burkhardt 2012) the Barrier operation has been characterised in simulation and in a small compute cluster.

8.6.1 Link Delay Measurements

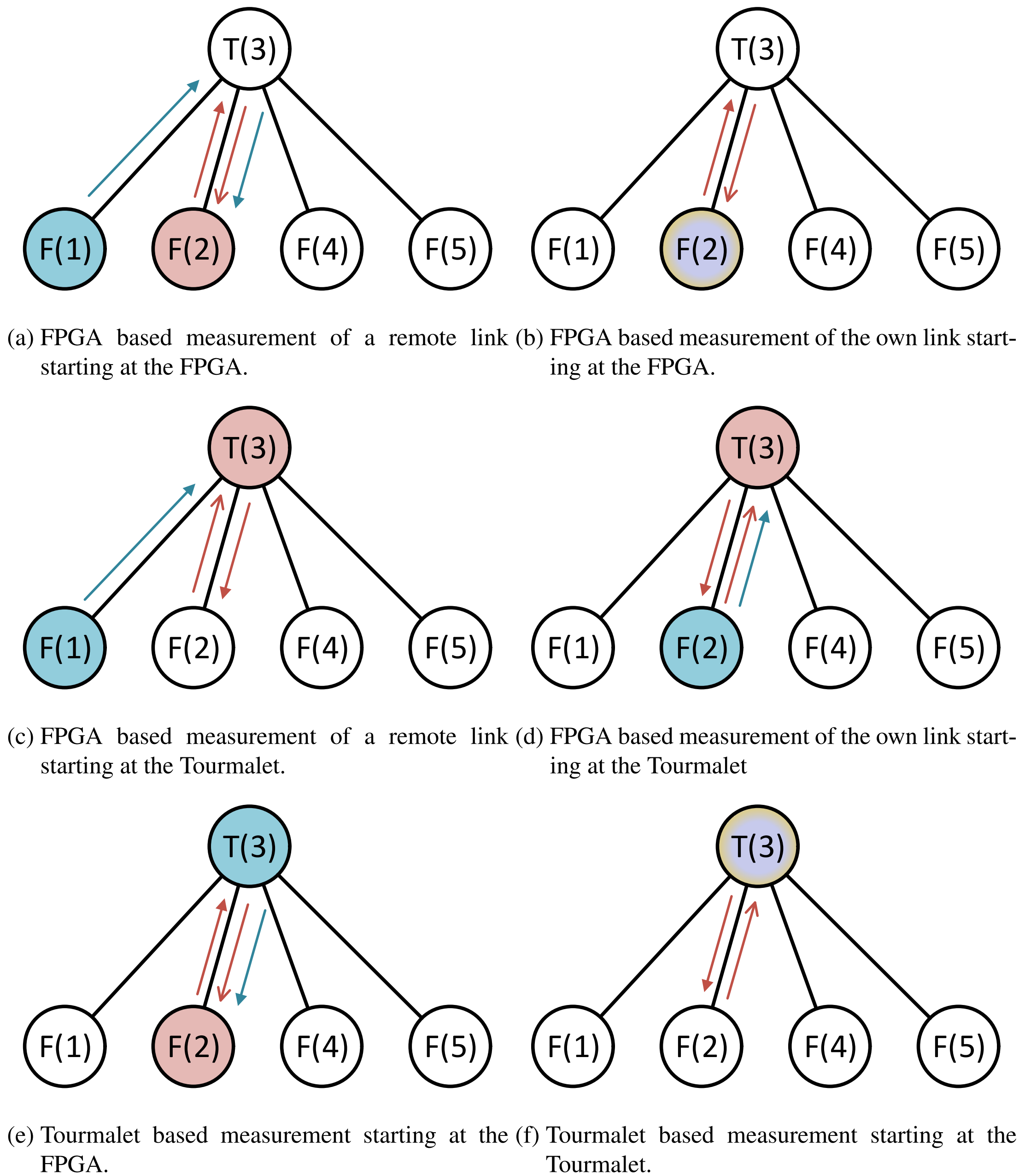
In order for the Interrupt operation to function as intended, the delay counter on every node has to be configured according to the actual link latencies between nodes in the network. For the precise measurement of these latencies, the Interrupt units implement a dedicated mechanism, measuring the round-trip delay of an Interrupt message on the respective link. With this mechanism, a timer on the initiating node starts counting clock cycles when the Interrupt message is created. The message is then sent on its usual path across the link and looped back at the other side's node. The counter on the source node is stopped, as soon as the circulated Interrupt message is received. In the first place, the measurement cycle is started by writing to a specific Registerfile address.

This round-trip measurement has been conducted for each link and each direction in a small EXTOLL network consisting of one Tourmalet ASIC and four BrainScaleS FPGAs. Figure 8.14 shows the network used topology and the different modalities that the round-trip latencies have been measured with. The measurements have been carried out, either mastered by a playback-program (cf. Section 3.2.2) on one of the FPGAs, or by a measurement script on the host-computer, where the Tourmalet card is mounted. During the measurements, each node (in the following referred to as *experiment master*) has mastered the measurement of each link and each link has been measured in both directions, i.e. once initiating the round-trip message at the local side and once at the remote side of the respective link. Results of these measurements are shown in Figure 8.15 and Figure 8.16. Thereby, for each link, all measurements from different experiment masters have been plotted on the same axes and separated by **blue** vertical dashed lines.

Because of the different clock frequencies on the Tourmalet ASIC (600 MHz) and the BrainScaleS FPGA (150 MHz, compare Section 8.2.2), the numbers of measured clock cycles on both node types differ by a factor of approximately four. The converted mean values in units of ns match in the range of one standard deviation.

Furthermore, it can be observed that if measuring the link starting at the FPGA side, the measured latency values behave less regular if the *experiment master* measures its own link (Figure 8.15, **M-F1** at link **1-3**, **M-F2** at link **2-3**, **M-F4** at link **4-3** and **M-F5** at link **5-3**) and more regular otherwise. If measuring the link from the Tourmalet side (Figure 8.16), this observation is inverted.

A possible explanation for this can be found when looking at the messaging patterns indicated with



(a) FPGA based measurement of a remote link starting at the FPGA.

(b) FPGA based measurement of the own link starting at the FPGA.

(c) FPGA based measurement of a remote link starting at the Tourmalet.

(d) FPGA based measurement of the own link starting at the Tourmalet

(e) Tourmalet based measurement starting at the FPGA.

(f) Tourmalet based measurement starting at the Tourmalet.

Figure 8.14: Different measurement modalities for the link round-trip latencies in the network. The experiment is executed on the **blue nodes** while the **red nodes** initiate the round-trip message.

coloured arrows in Figure 8.14. By comparing the measurement modalities shown in Figure 8.14 with the results in Figure 8.15 and Figure 8.16, one can see that the values behave more regular, always when the Registerfile write-message triggering the actual measurement (shown with a **blue arrow** in Figure 8.14) does pass the link to be measured before the measurement is started. This prepares the EXTOLL Link-Ports on either side of the link to be in the same state before every measurement and thereby stabilising the latency.

Notably, when the measurement is mastered by a script on the host-computer, the latency is always either regular or irregular for each link, depending on the start-side for the round-trip. This is because

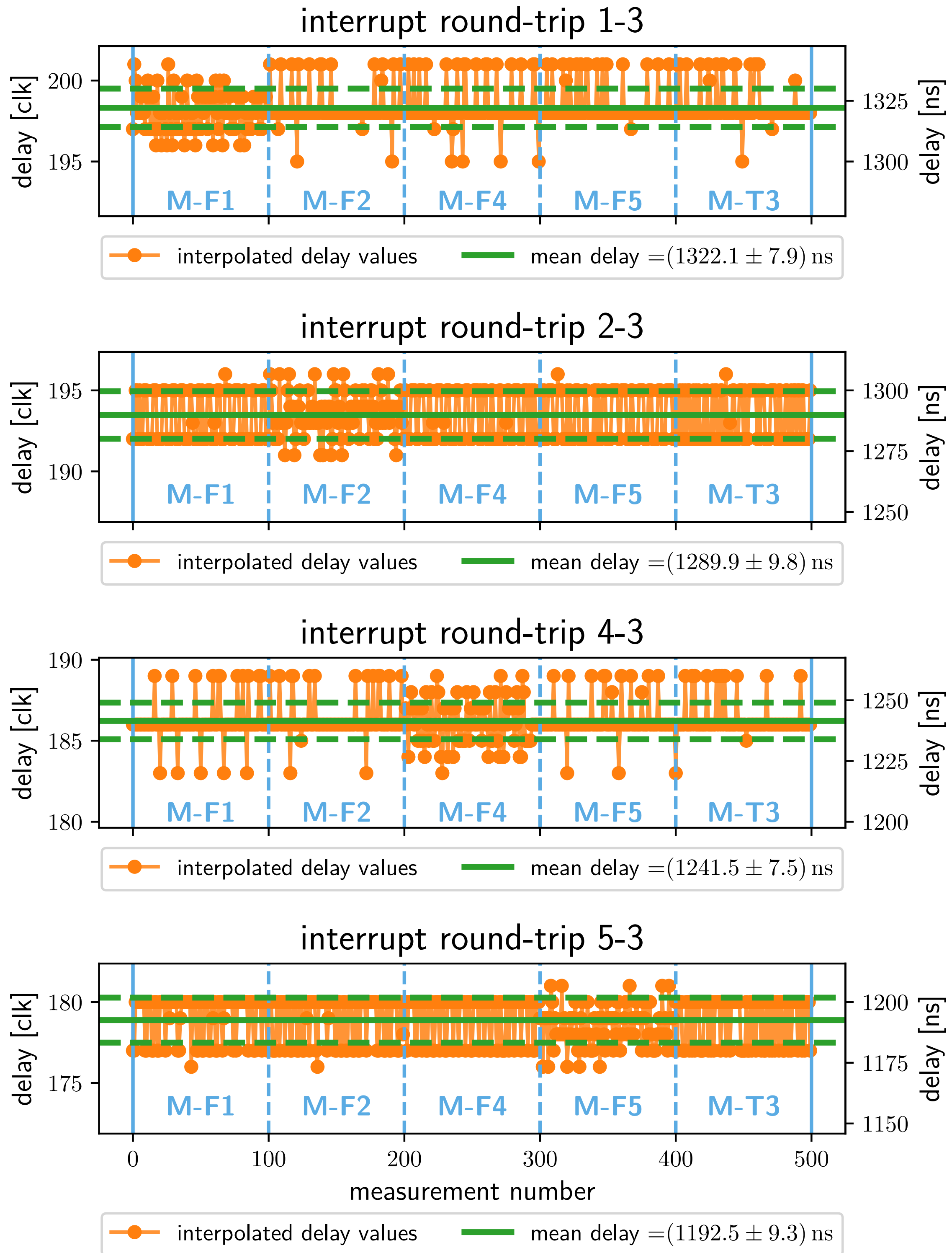


Figure 8.15: Results of the round-trip measurements starting at the link's FPGA side. Vertical lines indicate the border between different measurement masters, i.e. which node was used to configure the measurement. The respective master node is indicated by the blue text labels.

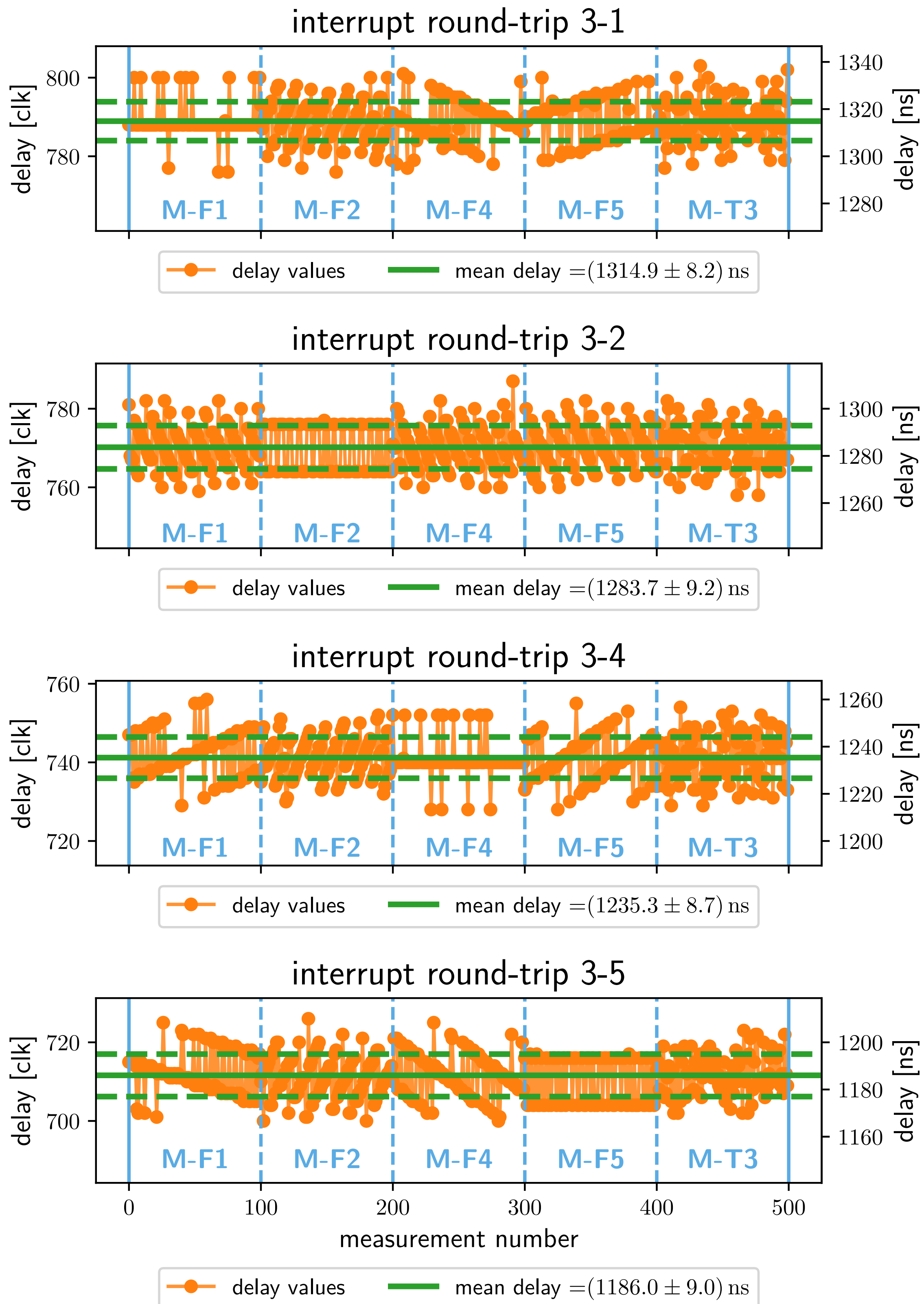


Figure 8.16: Results of the round-trip measurements starting at the link's Tourmalet side. Vertical lines indicate the border between different measurement masters, i.e. which node was used to configure the measurement. The respective master node is indicated by the blue text labels.

in this case every link is equal with respect to the Tourmalet node and the respective link will never be pre-conditioned when starting at the (local) Tourmalet and always will be pre-conditioned when starting at the (remote) FPGA.

Another observation is that the measured latency values are bimodal, i.e. mostly jump between two values separated by a constant interval. This interval is independent of the direction of measurement (20 ns, corresponding to 3 clk on an FPGA and 12 clk on the Tourmalet), which can be attributed to the round-trip nature of the measurement. This effect is most apparent in the cases where the latency is steadied by link preparation through the triggering Registerfile-write message. In cases with irregular latency values, where the links are not pre-conditioned, the values perform some drifting movement, while keeping the same constant interval. This can for example be clearly observed in Figure 8.16. In some cases, this pattern however is (partly) hidden away by sampling effects probably created by the repeat rate of measurements.

In order to fully explain this observed latency behaviour a deep insight into the hardware implementation of the link would be necessary. However, as the original purpose of these measurements is to obtain a typical delay value for configuring an interrupt synchronisation tree, these (systematic) latency variations are not of detailed importance. Instead, an averaged value plus standard deviation uncertainty of the measured round-trip latency will be used.

Hereby, it is important to note that this mean value does not experience a systematic drift on relevant timescales. This has been validated by performing these measurements at large intervals at the order of minutes up to days, yielding the same values. However, systematic offsets in the mean delay value have been observed after powercycling either the FPGA or the Tourmalet on either side of a link. This is exemplarily shown for the link between FPGA node 1 and Tourmalet node 3 in Figure 8.17. The reason for these systematic offsets is believed to be the repeated link training, finding a slightly different agreement on the mutual parameters.

Because of this powercycle effect, it was decided to always repeat a fresh measurement cycle of all link latencies and setting up a new Interrupt tree before performing an experiment across multiple BSS-2 FPGAs.

8.6.2 Interrupt- and Barrier Tree Generation

Having measured the round-trip latencies for each link in the network, the task is now to derive delay values in units of clock cycles for each node that is part of the interrupt-group which is to be organised in a virtual tree topology. The tree structure is thereby configured in the Registerfile of the Interrupt units by storing the link-numbers towards the respective node's children and towards its parent node. The derivation of the target delay values for each node is illustrated in Figure 8.18. Algorithmically, the derivation works as follows:

1. Find the longest distance in terms of link-delays (in time-units) from the root node to a leaf node while constructing the tree.

2. Set the delay value for the root node to the total latency of the longest path plus a small offset. The offset is required, as for technical reasons, the interrupt delay must at least be one clock cycle at every node.

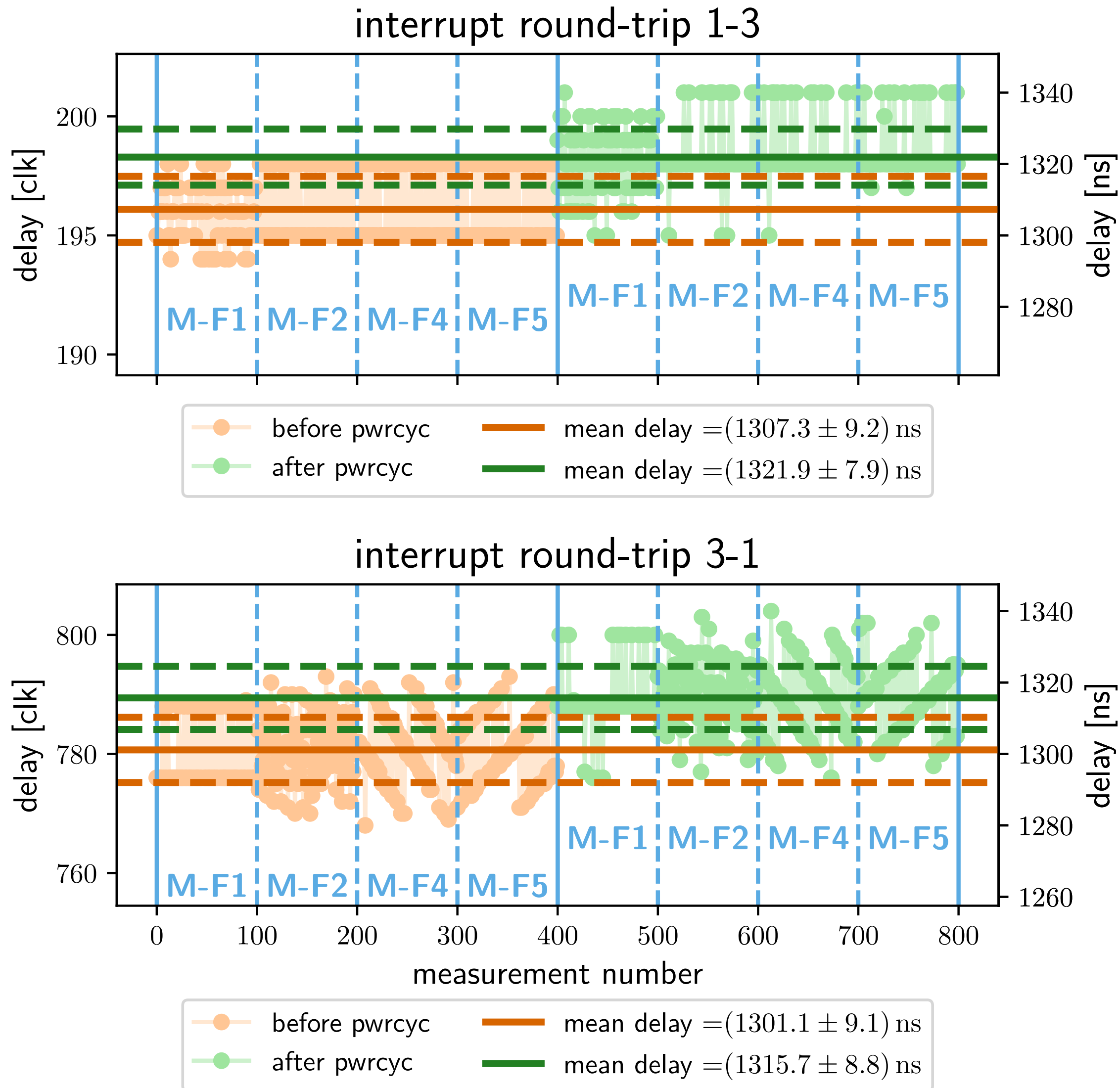


Figure 8.17: Measurement of the round-trip latency before and after a powercycle of a node involved in the measured link. The mean values differ systematically.

3. For each node in the tree, starting at the root, subtract the link latency from the parent node of the parent's delay value. Set this as the respective node's delay value.

4. Convert the delay values from time-units to clock cycles using the respective node's operation frequency.

It should be noted that as the nodes are generally operating at different clock speeds, it is important to perform the calculations in common absolute time units and only convert the result to clock cycle units relative to the respective clock frequency afterwards. Following this constraint, the slowest clock frequency in the network determines the offset value as the slower nodes cannot wait a fractional number of cycles for balanced delays.

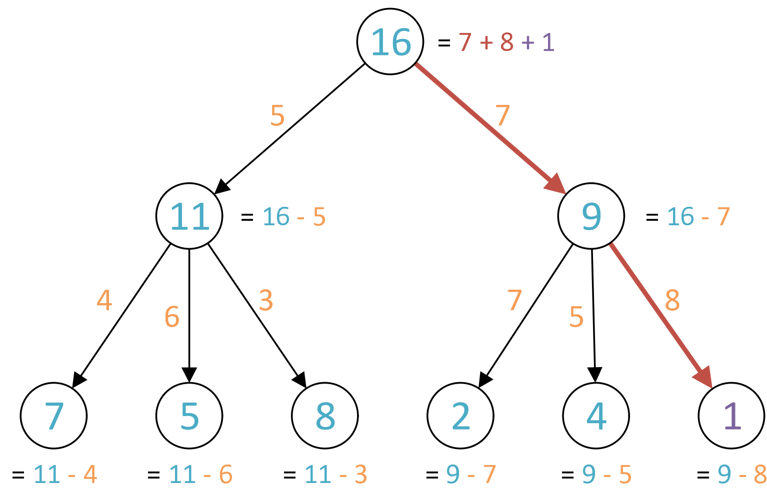


Figure 8.18: Exemplary derivation of **interrupt delays** from **known link latencies** in an interrupt tree. The **longest leaf-path** serves as basis for the overall global interrupt delay. For technical reasons, the delay has to be **at least one cycle**.

8.6.3 Interrupt Synchrony Measurements

Now, after having described the procedures of link delay measurements and creation of an interrupt tree, these shall be used in order to measure the actual synchrony of the Interrupt signal arriving at the different FPGAs.
The experiment protocol is defined as follows:

1. a) Measure the latencies of every link from a specific FPGA as playback master.
   b) Repeat step 1a 100 times in a single playback program with minimal time in between.
   c) Repeat the measurement playback program from step 1b 3 times.
   d) Calculate the average delay for every link.
2. Build an interrupt tree using the delays from the previous step 1 using a specific node as tree root.
3. Capture the time, when the interrupt signal is registered at the FPGA logic using an Oscilloscope attached to GPIO pins connected to the interrupt signal.
4. Repeat steps 1 to 3 300 times with 0.5 s time in between.
5. Repeat step 4 for every combination of playback master and root node.

An exemplary screenshot of a step 3 oscilloscope measurement is presented in Figure 8.19. The channel traces respectively depict the interrupt signal at each FPGA node, lasting for a single clock cycle. The oscilloscope has been configured to automatically measure the mutual skew between the four signals at their respective rising edge onset point.
Figure 8.20a shows the raw results of these measurements after step 4. The measured skews are thereby, without loss of generality all plotted relative to node N1 for comparability. One can observe that the individual traces always jump between values that are separated by a clock period of 8 ns.

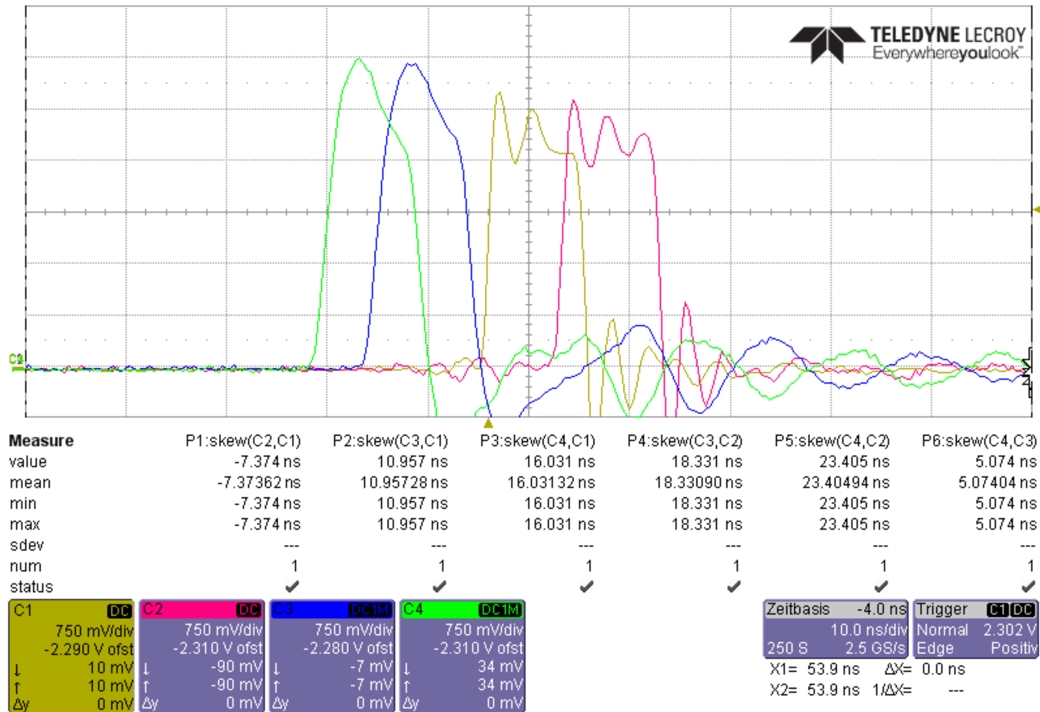
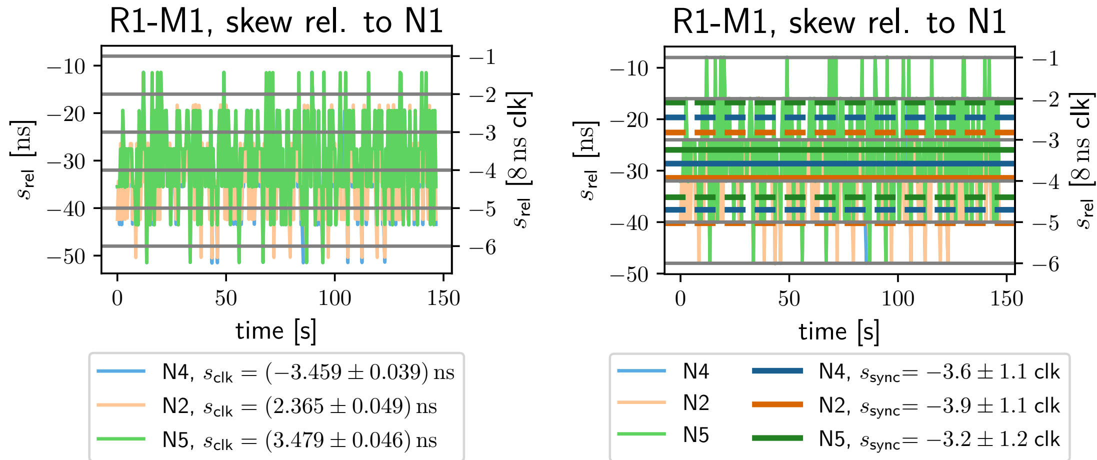


Figure 8.19: Oscilloscope screenshot of the interrupt synchrony measurement (day A). Channels translate to node-ids as follows: C1 - N4, C2 - N1, C3 - N2, C4 - N5.



(a) Raw data with offsets due to clk-skew (s_{clk}) between FPGAs.

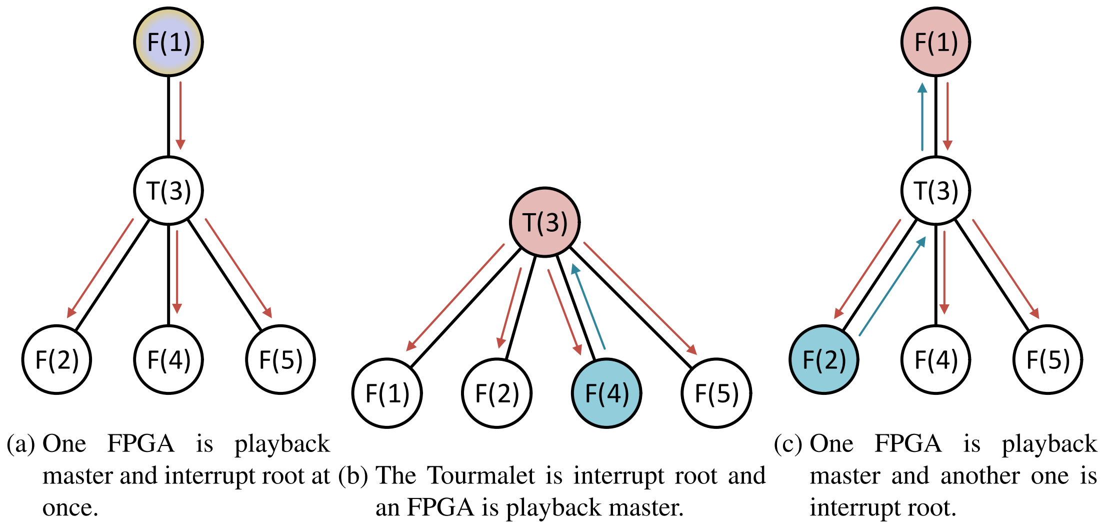
(b) Corrected synchronisation-skew (s_{sync}) between FPGAs with extracted mean values and uncertainties.

Figure 8.20: Postprocessing and extraction of raw measurement data for the relative interrupt skew between FPGAs.

However, the skew between distinct FPGAs traces is not aligned to multiples of a clock period. This is due to the fact that the FPGAs do not operate on a synchronised clock signal, but rather exhibit an intrinsic clock skew that can be extracted from these measurements by rounding the raw values to the next 8 ns *grid position*.

After correcting the measured skew values with respect to the respectively extracted clk-skew, the pure interrupt synchronisation skew which is the goal of these measurements can be determined

in units of clock cycles, as shown in Figure 8.20b. The skew has been averaged for each node and marked in Figure 8.20b with horizontal lines for the mean value and the respective standard uncertainty.



(a) One FPGA is playback master and interrupt root at once.

(b) The Tourmalet is interrupt root and an FPGA is playback master.

(c) One FPGA is playback master and another one is interrupt root.

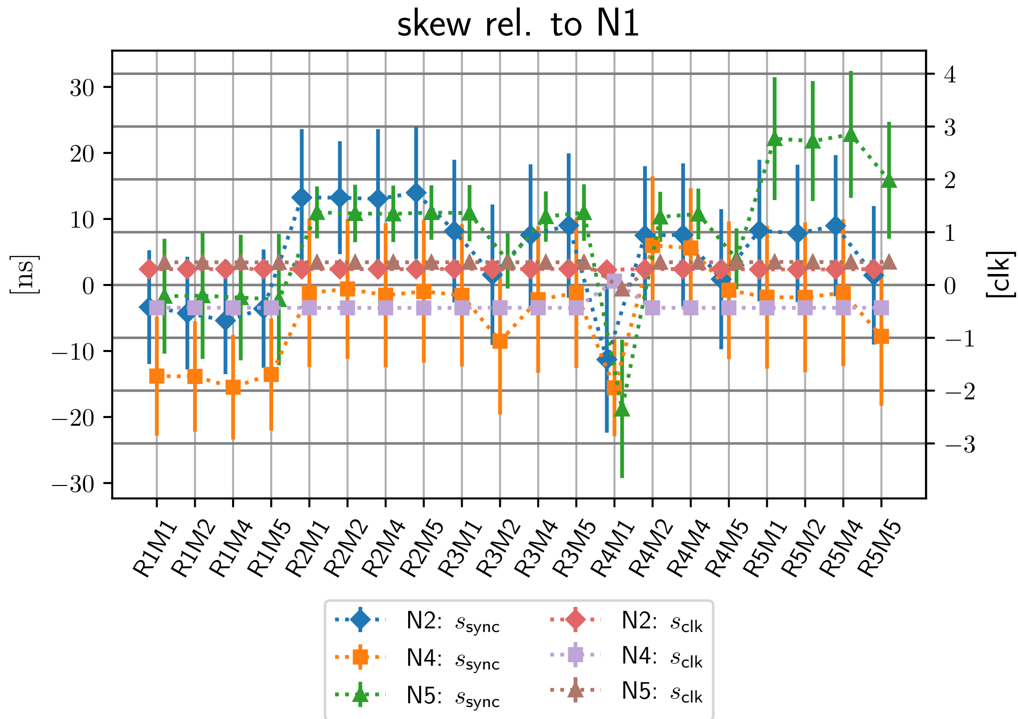
Figure 8.21: Different modalities for the Interrupt operation in the prototype network.

This is now repeated for every possible interrupt operation modality according to step 5. Figure 8.21 shows the principally distinguished modalities according to Section 8.6.1 and Section 8.6.2. In the small prototype network used in this work, the interrupt tree can have two levels at maximum below the root, if one of the FPGAs is configured as root of the interrupt tree. If the Tourmalet ASIC is configured as root, the tree has only a single level. The other distinction criterion is, which FPGA node acts as playback master, as already discussed in Section 8.6.1. This determines, whether some (and which) links are pre-conditioned for the latency measurement and the interrupt operation. As the latency measurement is averaged over a total of 300 iterations in step 1, the changed irregularity of the measured latency should not have an impact on the interrupt operation. However the links will be (un-) pre-conditioned also in the actual interrupt operation based on which node is to be the experiment master, which will probably have an impact on the interrupt accuracy.

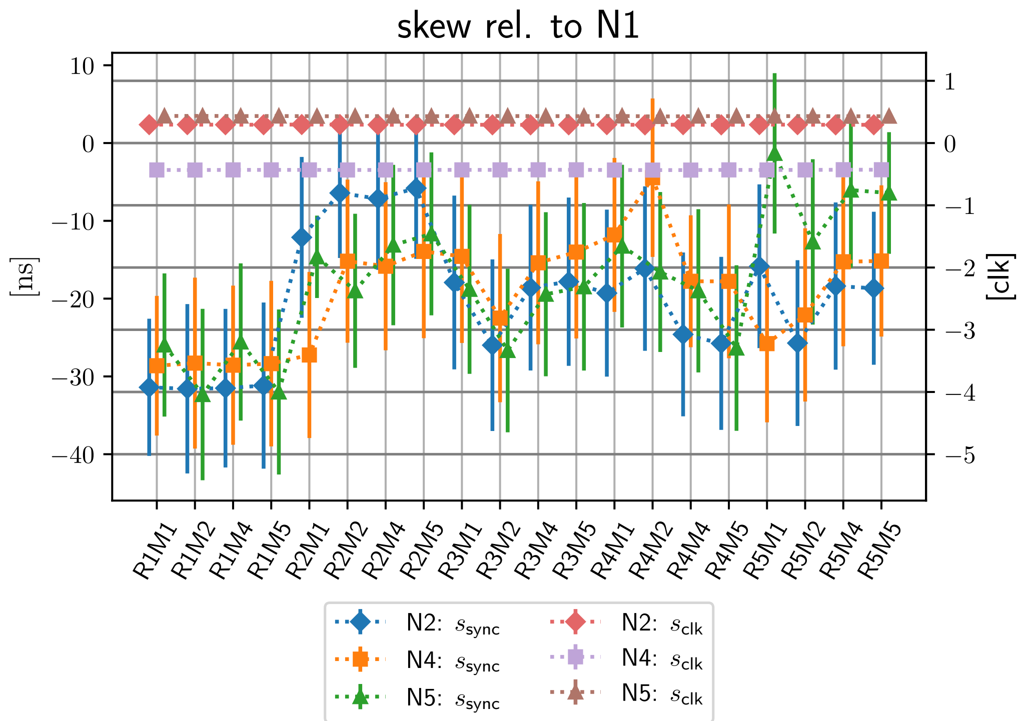
The overall results of the step 5 are shown in Figure 8.22. In order to assess the overall stability of the measured observations, the measurements have been repeated twice with a time distance of 5 days, shown in Figure 8.22a and 8.22b. Every data point in these two plots shows an average skew value together with its associated uncertainty error-bar.

The first important observation is that the extracted clock-skew s_{clk} is constant and very precise, as all values are the same on both days and for every measurement modality. Additionally, the error-bars are so small that they are not perceivable in the plots. Overall, there is only a single outlier for `R4M1`, where also the synchronisation skew s_{sync} shows a clear outlier at every node.

From the previously made link pre-conditioning considerations, one would expect that the FPGA node acting as interrupt root would exhibit a higher skew with respect to the other FPGA nodes. This is considered to be the case, as by notifying the root to start the interrupt operation, the root's link is pre-conditioned. From these considerations, one would also expect that in case of the root node also being the master, no link should be pre-conditioned counteracting the previous effect.



(a) Measurements on day A.



(b) Measurements on day B.

Figure 8.22: Collected mean values of clock-skew and synchronisation-skew for all measured modalities.

Another expected effect on the measured skew arises from the fact that a root FPGA node is two layers apart from the other FPGAs in the interrupt tree. This is expected to multiply the calibration

inaccuracies and therefore also effect the relative skew.
Regarding the results shown in Figure 8.22, one should pay attention to the fact that all values are plotted relative to the node **N1**. Therefore relative skews between other node pairs have to be read from the position of those data points relative to each other, rather than their absolute position on the vertical axis.
When looking at the results, it can be clearly observed that the respective root node exhibits a larger absolute value relative to **N1** and also a larger distance relative to the other nodes. For a concrete result-examination, in Figure 8.22a, the values of **N2** rise above **N4** and **N5** when it is configured root, while **N5** rises above the two others accordingly. Also, **N4** rises relative to the others accordingly although it does not cross their lines due to its lower relative starting point. Furthermore, also for **N1** being the root, all other nodes' skew values drop in their absolute axis-value, resembling a rise of **N1** as their node of reference. This can also be confirmed on the second day measurement in Figure 8.22b.
However, the counteracting effect expected for root-nodes also acting as experiment master cannot clearly be observed. It is therefore inferred that the distance from the root to the leafs has a larger effect, than the link pre-conditioning.
Altogether, it should be noted that all presented values jump around a lot, which is manifested in the error-bars in both plots being rather large with respect to the previously described trends. Also it is largely possible for single nodes to exhibit persistent offsets, as can be seen in Figure 8.22b, where all measured skew values are negative relative to **N1**, but otherwise exhibit the same patterns as in Figure 8.22a.
In summary, it can be found that the interrupt operation is precise with an overall inter-FPGA-uncertainty u_{int} of approximately up to 5 8 ns clock cycles in both directions in the worst case

$$u_{\text{int}} \approx \pm 5\,\text{clk} = \pm 40\,\text{ns} \quad . \tag{8.11}$$

However, it should be noted that this result has been obtained in a relatively small prototype network and might not be valid in a larger network. For example it could be that in a larger network with a deeper interrupt tree, the differences in the skew, depending on the root node were even larger. In that case there are several methods of mitigation towards this effect. First, under the assumption that the observed effect is predictable with respect to the tree-distance, one could embed a countermeasure into the delay calibration. Another possibility would be to arrange the network in such a way that all FPGAs are leaf nodes at the same level and operate a Tourmalet node as root. This would be effective except for the link pre-conditioning effect that could not be observed in the prototype network.

8.6.4 Synchronous Experiment Execution

On the topmost software level, experiments are defined and executed using a modelling wrapper like *PyNN* (Davison et al. 2009; Electronic Visions(s), Heidelberg University n.d.[j]) or *HxTorch* (Electronic Visions(s), Heidelberg University n.d.[h]; Paszke et al. 2019; Spilger et al. 2023). Up to recently, the BrainScaleS-2 experiment flow involved only the execution of single-chip experiments by connecting to a specific single BSS-2 FPGA and transferring a playback program for execution. In the work of (Straub 2023), an *HxTorch* experiment has been partitioned into multiple single-chip

executions that can be run sequentially on the same chip. In continuation of this work it will also be possible to run these executions independently in parallel on multiple chips, depending on the partitioning structure. What is currently missing, is the possibility to execute an experiment across multiple BSS-2 ASICs while exchanging spike events in real time between them. Here, the focus shall be on the *PyNN* wrapper, but in principle the following synchronisation strategy should also be applicable to other modelling wrappers like *HxTorch*.
As summarised in Section 3.3, a PyNN experiment runs on a global simulator state which is initialised be a setup-call and the creation of neuron populations and projections between them before it can be executed and finally evaluated. Without non-trivial changes to the underlying *grenade* experiment description layer, the *PyNN simulator* will control a single pair of BSS-2 FPGA and HICANN-X ASIC.
In order to run a neuromorphic experiment across several BSS-2 ASICs, measures have to be taken to partition the experiment execution between the systems. First, the systems have to be configured for sending outgoing spike events to the correct remote system, where the receiving neurons of the inter-chip projections are placed. Second, the systimes and execution flow of playback programs have to be synchronised in order to make the spike-times compatible across inter-chip projections.
To achieve the former goal, the mapped output spike-labels of the source population and the input synapse-labels of the destination populations (cf. Section 3.1.1) have to be retrieved from the *grenade* data structures. Therefore, the neuron populations need to be defined and processed by *grenade* for mapping to hardware spike-labels. This process is triggered with a *dummy-execution* of the *PyNN simulator* by calling `pynn.run(None)` without specifying an experiment execution time. After having retrieved the labels, they have to be configured to the FPGAs' Destination Mapping lookup-tables together with the EXTOLL node-id of the destination FPGA (cf. Section 7.3.2).
Due to the global *simulator state* of *PyNN* and the fact that python internally serialises the execution of threads with a Global Interpreter Lock (GIL) (Eggen et al. 2019), it is not possible to run multiple *PyNN* instances in parallel using threads. Instead, one has to partition the execution into several processes, each possessing an own python interpreter. This significantly complicates the exchange of information, like the receiving synapse labels, between those instances. Consequently, the input synapse labels of one experiment partition have to be communicated to the process of that partition emitting spikes to be sent to those synapses.
Finally, the goal of synchronising the execution flow of playback programs on multiple FPGAs and PyNN processes, can be achieved by using the playback instructions for Barrier and Interrupt operations (cf. Section 4.1.2 and Section 7.2.2, as well as Section 8.6.2).
The preparation and synchronisation procedure thereby involves the following steps:

1. Measure and configure the interrupt- and barrier tree according to Section 8.6.1 and Section 8.6.2.

2. Create a new process for each BSS-2 FPGA-ASIC pair taking part in the experiment. Mark one process as experiment master. For each process:

    a) Define the part of the neural network to be emulated on this chip.

    b) Retrieve the neuron- and synapse labels and make the latter available to the other processes, e.g. using methods of shared memory.

c) Configure the lookup-tables in the Destination Mapping unit of the FPGA local to the respective process using the local emitting neuron labels and the remote receiving synapse labels.

d) During experiment execution via injected configurations (cf. Section 3.3):

- pre-realtime:
  - Arm the interrupt receivers in the *Playback Executor* for the global-interrupt barrier-wait-operation (cf. Section 3.2.2) and the *Event Switch* for storing the global systime offset (cf. Section 7.2.2).
  - Perform a global Barrier operation and block playback execution until all FPGAs have reached this point.
- inside-realtime-begin:
  - Configure the *Event Switch* to forward events from the ASIC to the Event Communication units (cf. Section 7.1).
  - Perform a global Barrier operation.
  - If this process is the experiment master, trigger a global Interrupt operation at the root node of the configured interrupt tree.
  - Wait until the Interrupt message has arrived. Now the systime is synchronised and the systime offset is stored for timestamp conversions (cf. Section 7.2.2).
- inside-realtime-end:
  - Configure the *Event Switch* to not forward events from the ASIC to the Event Communication units anymore.
  - Perform a global Barrier operation.
- post-realtime:
  - Dis-arm the interrupt receivers.

e) Clean up the lookup-tables by invalidating the previously written entries.

f) Retrieve the experiment results (recorded spike trains and Analog to Digital Converter (ADC) traces) and make them available to the original process.

3. Evaluate all the results after all processes have finished their execution.

8.7 The Synfire Chain Experiment

As a first multi-chip neuromorphic experiment, a Synfire Chain model was implemented across two BSS-2 systems. Historically, the Synfire Chain model has been inspired by (Abeles 1991; Aertsen et al. 1996; Diesmann et al. 1999; Gewaltig et al. 2001) and investigated with respect to its neural signal propagation properties by (Kremkow et al. 2010). The latter work shows that this model "increases the selectivity for propagation of synchrony through a feedforward network" and cites this behaviour to be also observed in biological in vivo studies.

Generally, a Synfire Chain consists of multiple connected chain links, each consisting of an excitatory and an inhibitory population of neurons. While the excitatory populations actively carry the feedforward signal through the chain, the inhibitory populations mediate the activity of the excitatory ones by being stimulated by the same feedforward signal, and inhibiting the excitatory population in their local chain link. A depiction of this neural network topology is shown in Figure 8.23, where the **excitatory populations** are coloured in red and the **inhibitory populations** are coloured in blue.

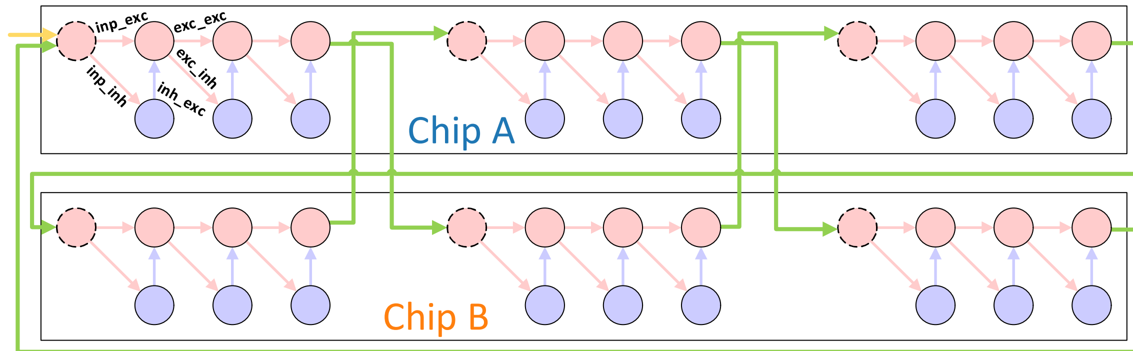


Figure 8.23: Population projection graph of a Synfire Chain network spanning two BSS-2 ASICs. The **excitatory populations** will excite all neurons at the next chain link, while the **inhibitory populations** will inhibit the excitatory neurons at their own chain link. This synfire chain is broken into several parts which are connected back and forth between the two chips. Activity is started through a stimulus projection at an excitatory input population on **Chip A**. This figure is modified from (Müller, Emmel, et al. 2023) where it has been contributed by the author of this thesis.

For benchmarking purposes, the Synfire Chain is connected back and forth between two synchronised BSS-2 systems. Thereby, each break in the chain collects spikes from an excitatory population and transfers them to the retrieved event-label addresses of an excitatory input population (boldly dashed in Figure 8.23) on the remote chip, as explained in Section 8.6.4 and using the spike event communication architecture described in Chapter 7. Stimulus input is injected at least at the first chain link, but can be configured to be added at any chain break input population. Likewise, the number of chain links and breaks can be configured for the experiment. The projection weights, named as indicated in Figure 8.23 where chosen manually in order to obtain a well propagating signal. The chosen values in a range between $[1,63]$ are listed in Table 8.1. Thereby the value for the $w_{\text{inh-exc}}$ weight is implemented as a free parameter, defaulting to -50.

Projection	Weight
$w_{\text{inp-exc}}$	60
$w_{\text{inp-inh}}$	40
$w_{\text{exc-exc}}$	50
$w_{\text{exc-inh}}$	30
$w_{\text{inh-exc}}$	-X = -50

Table 8.1: Chosen weight values for the Synfire Chain implementation. The $w_{\text{inh-exc}}$ value is varied with a default of -50.

The population sizes in the chain links respectively comprise 7 excitatory and inhibitory neurons.

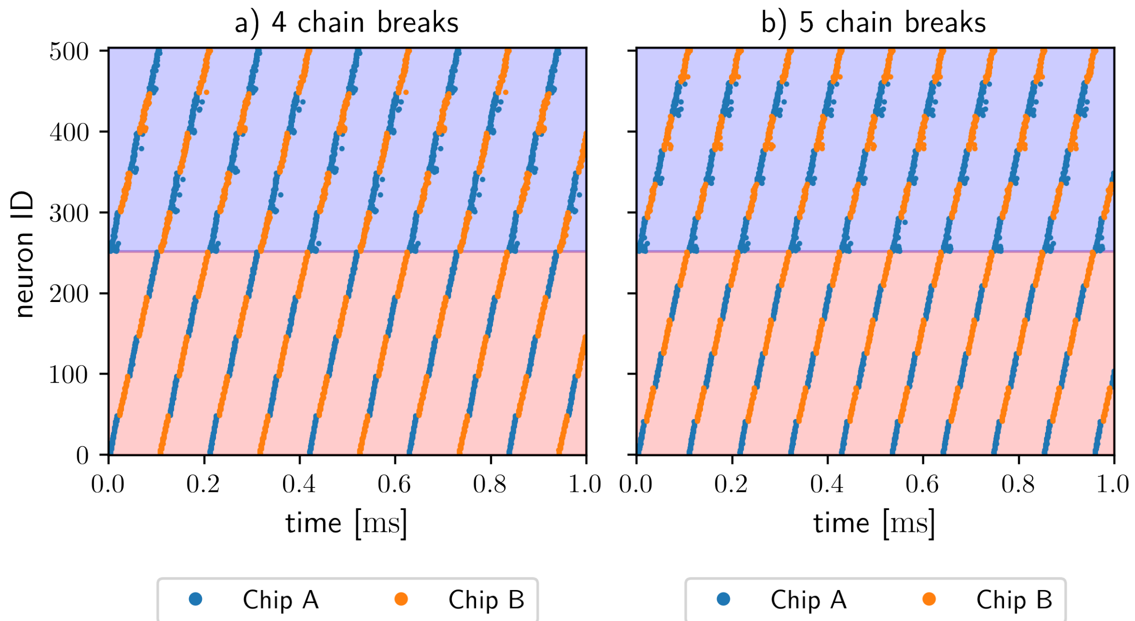


Figure 8.24: Resulting Synfire Chain activity diagrams of the synfire chain experiment with even (a) and odd (b) number of chain breaks at the excitatory projections are presented here. The neuron ids are numbered independently on both chips, and the respective spikes are plotted on the same axis using different colours. **Inhibitory spike trains** are plotted at the top while **excitatory spike trains** are plotted at the bottom half. The time axis is given in hardware units. Full participation of all neurons in the Synfire Chain can only be reached with an even number of chain breaks.

Figure 8.24 shows the resulting spike-trains from both chips with a number of 36 chain links per chip, as this is the highest number implementable at the given population size and a total of 512 available neurons on each chip, using 504 neurons on each chip. On the left plot (a), the chain has been broken 4 times, while it has been broken apart 5 times on the right plot (b). Due to the geometry of the described topology (cf. Figure 8.23), only with an even number of chain breaks, the whole chain will take part in the signal propagation. This can be seen in Figure 8.24 where on the right plot (b), the neurons of the respective chip repeat their activation pattern at every cycle, while on the left plot (a) they alternate.

Generally one can also observe that the inhibitory activation patterns are not as neat, as the excitatory ones. This is very well expected as the inhibitory neurons moderate the excitatory ones, but not themselves.

Figure 8.25 shows results obtained from changing the inhibitory weight $w_{\text{inh-exc}}$ from the default (-50, cf. Figure 8.24) to lower absolute values. When the inhibitory weight is lowered too much, the signal-transmission starts blurring out (cf. Figure 8.25 (c) and (d)). It can be observed that the number of events, accumulated per packet is quite low between 1.9 $\frac{\text{evt}}{\text{pkt}}$ and 2.3 $\frac{\text{evt}}{\text{pkt}}$. This is because in the Synfire Chain model exhibits serial activity rather than simultaneous activity. At the bottom right plot (d) of Figure 8.25 one can observe a little higher number of events per packet, as more activity is generated. However, this is still quite a small number and illustrates that even as the plot may look like simultaneous spiking, the events are still quite sequential with an Inter Spike Interval of the merged spike trains being larger than the default value for the packet timeout which is set to 50

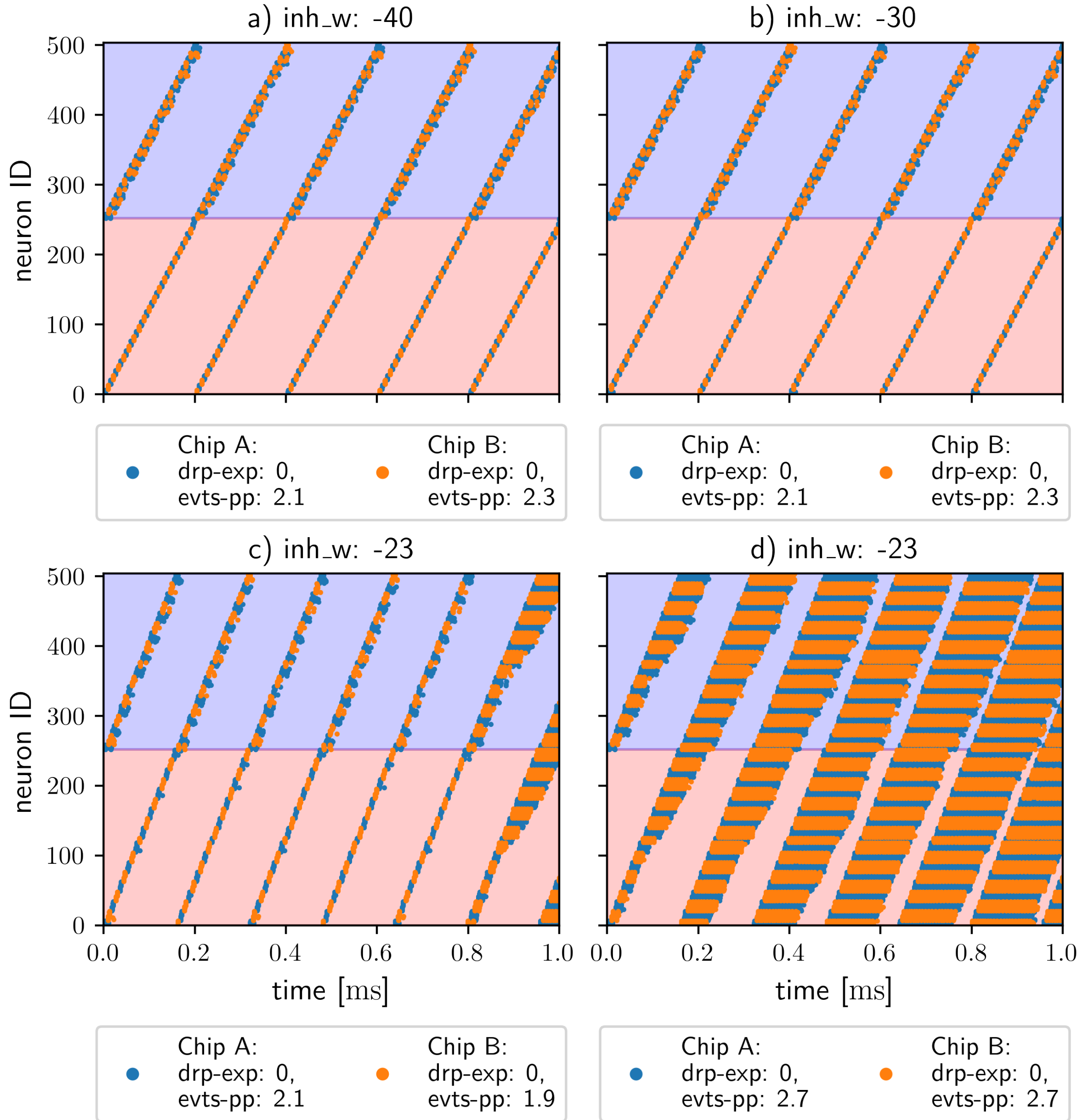


Figure 8.25: Resulting Synfire Chain activity diagrams when lowering the absolute value of the inhibitory weight $w_{\text{inh-exc}}$. The Figure legends additionally show additional information about the number of dropped events due to expired timestamps (*drp-exp*) and the number of events per network packet (*evts-pp*).

clock cycles. Furthermore, as Figure 8.25 (c) and (d) where both recorded with the same inhibitory weight setting, the point in time when the Synfire Chain starts dispersing is somewhat random or rather sensitively depending on small fluctuations in the activity rate at individual chain links.

Figure 8.26 shows experiment results obtained from changing the value of the modelled axonal delay, as well as the timeout value for the accumulation of spike events to packets (cf. Section 7.3.3). It is observed that the number of dropped events due to expired timestamp rises quickly when the axonal delay is less than 250 clock cycles higher than the programmed timeout value (Figure 8.26 (a) to (c)). In Figure 8.26 (a) the drop rate is even large enough to break the signal propagation through the Synfire Chain, which is why the absolute drop number lowers in this case. With Equation (5.4), this leeds to the conclusion that the total transmission delay through the network, including the two

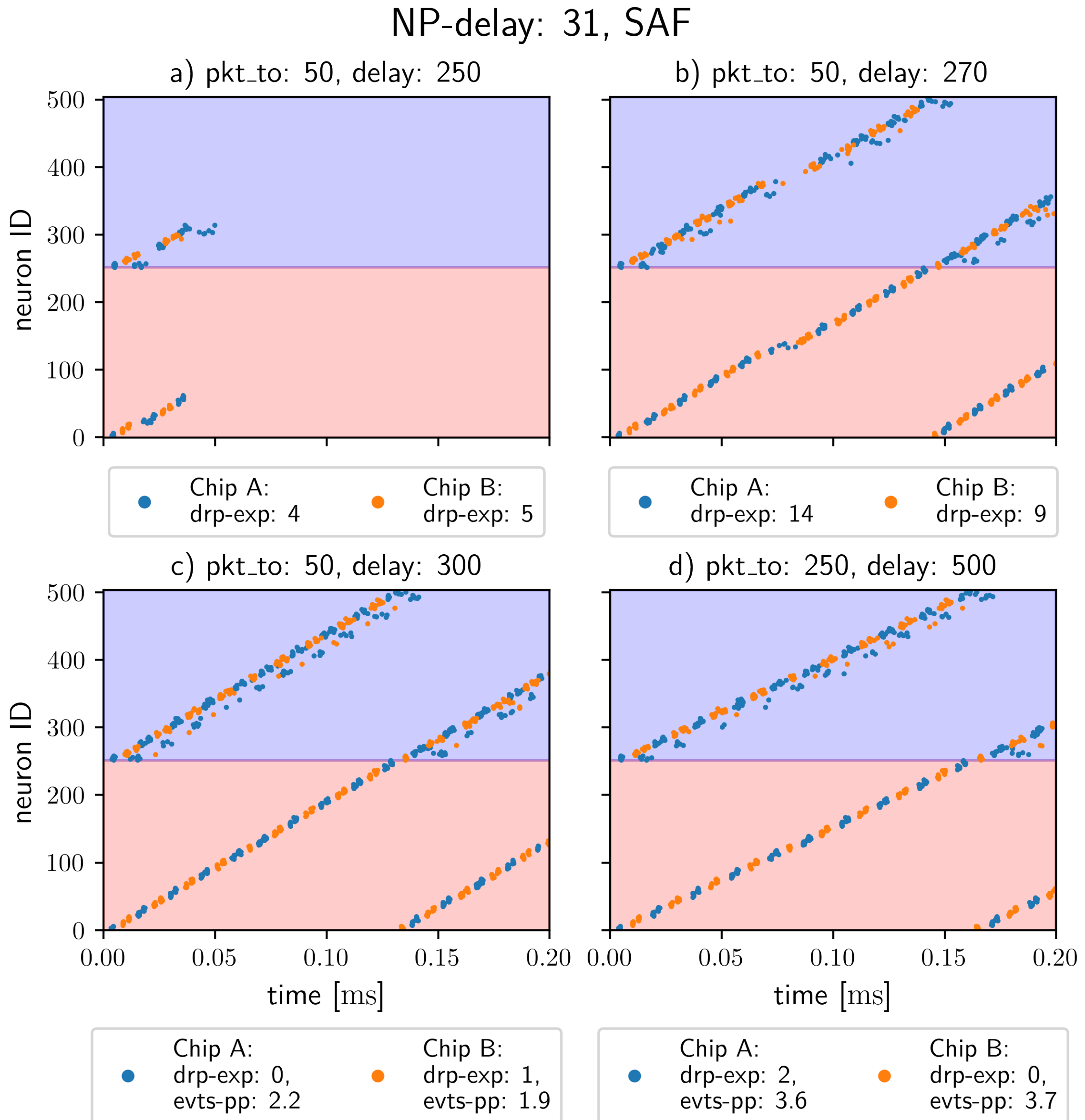


Figure 8.26: Resulting Synfire Chain activity diagrams when changing the modelled axonal delay value for the inter-chip connections ((a) to (c)) as well as the packet aggregation timeout value (d). The Figure legends additionally show additional information about the number of dropped events due to expired timestamps (*drp-exp*) and the number of events per network packet (*evts-pp*). These plots have been recorded with unoptimised FPGA NP settings in SAF mode.

chip-links, must be upper-bounded in the order of 250 clock cycles, corresponding to 2 µs. This finding roughly agrees with the latency measurements in Section 8.5. The effect of increasing the axonal delay can also be observed directly in the plots at the larger gaps at the inter-chip chain breaks.

When increasing the packet aggregation timeout in Figure 8.26 (d) together with the delay, also an increased number of now between 3.6 $\frac{\text{evt}}{\text{pkt}}$ and 3.7 $\frac{\text{evt}}{\text{pkt}}$ can be observed. This is however still less than the expected maximum of 7 $\frac{\text{evt}}{\text{pkt}}$ according to the population size. This shows that the excitatory spike events in the chain links must be spread across more than approximately 2 µs in their time of arrival

at the accumulation bucket, as on average two packets are needed to transport them at a timeout value of 250 8 ns clock cycles. This is despite that according to the spike train's timestamp data, the activity is actually spread across approximately 1 μs, as can be seen in Figure 8.27 and therefore indicates a transmission spread of around 1 μs, probably introduced by the transmission from the chip. One probable reason for this is the additional transmission of lots of MADC data and spike events that are not transmitted across the network.

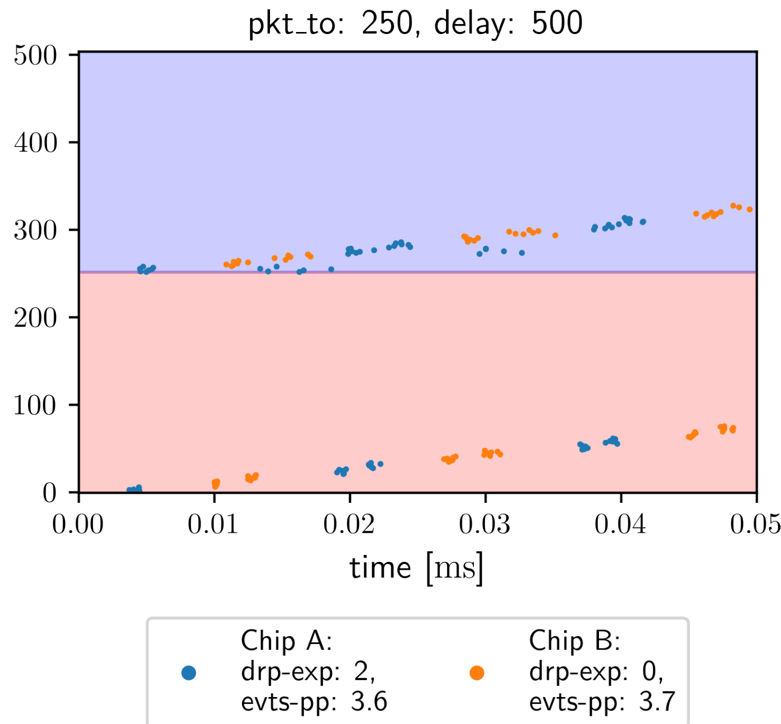


Figure 8.27: Zoom into Figure 8.26 (d).

It should be noted that the average value for the number of events per packet is calculated from a total count of events and a total count of packets, retrieved from performance counter registers in the FPGA, instead of having access to the actual packet sizes.
As described and measured in Section 8.5, the network latency can be optimised by configuring the FPGA's NP to operate in VCT mode instead of SAF and by reducing the number of words required in the NP's buffer to inject the first flit into the network to a minimum (the default value is 31 here). This optimisation leads to the results shown in Figure 8.28.
Now it can be observed that approximately the same number of expired-timestamp-drops occur for a lower axonal delay setting. From this finding one can deduce the network latency to be reduced by approximately $30\,\mathrm{clk} \cdot 8\,\frac{\mathrm{ns}}{\mathrm{clk}} = 240\,\mathrm{ns}$ which matches the mean value improvement found in Section 8.5.

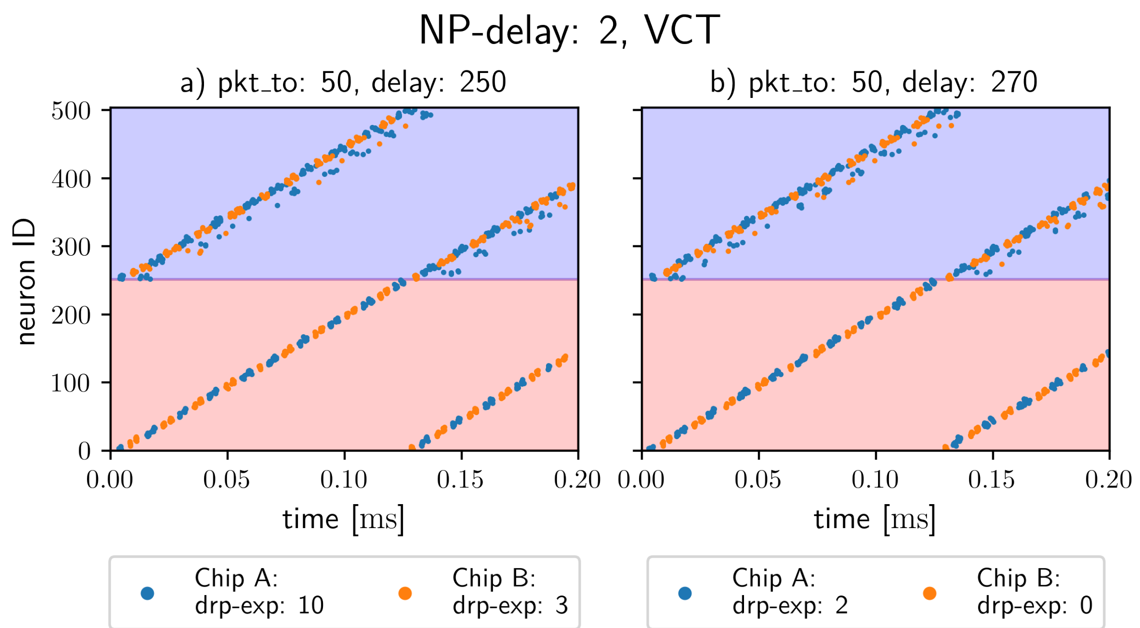


Figure 8.28: Resulting Synfire Chain activity diagrams when changing the modelled axonal delay value. The Figure legends additionally show the number of dropped events due to expired timestamps (*drp-exp*). These plots have been recorded with optimised FPGA NP settings in VCT mode.

9 Conclusion

Finally this thesis will be concluded by summarising the presented achievements and discussing the limitations of the presented implementation and how these can be improved in the future.

9.1 Summary

At the beginning of this thesis, an concise introduction is given on the background of neural information processing in biology (Section 2.1), neuromorphic computing in general (Section 2.2), as well as the principles and techniques of high-performance interconnection networks (Section 2.3), which are essential for both *massively parallel classical computers* and *large-scale neuromorphic computing systems*. Neuromorphic computing has been emerging as a specialised computational approach to machine learning and artificial intelligence in the last decades.

The main contribution of this thesis is the design and implementation of a packet-based event communication architecture for the BrainScaleS-2 neuromorphic computing system, using the EXTOLL interconnection network technology. Both, the BrainScaleS-2 system (Chapter 3) as well as the EXTOLL network technology (Chapter 4) have been introduced in detail. While BrainScaleS is a mixed-signal accelerated neuromorphic computing system which emulates neural dynamics using analogue electronics with a high speedup factor compared to biological timescales, the EXTOLL network provides the high bandwidth, as well as the low latency and high sustained injection rates of small packets, that is required to communicate neuromorphic spike events on those accelerated timescales. Another vital advantage of the EXTOLL network is the relatively easy access to the required network interface hardware units for an FPGA implementation, as the EXTOLL technology has been developed by the former *Computer Architecture Group* (CAG) within the *Institute of Computer Engineering* at University of Heidelberg and the EXTOLL company is a spin-off from CAG. This facilitated direct personal communication and support from the responsible experts.
Chapter 5 gives an overview on the concept of packet-based event communication for neuromorphic computing and puts it into context with respect to requirements regarding Quality of Service (QoS). The concept is also compared to existing neuromorphic computing systems from the historical background of the BrainScaleS-2 system, as well as the SpiNNaker system which has been developed in Manchester, parallel to the BrainScaleS system within the Human Brain Project, funded by the European Union.
To summarise, spike communication has to provide stable transmission latencies between the emulated neurons and synapses, which are placed on different chips, where the delay values can be user-defined through the axonal delays in the neural network model implemented on the neuromorphic computing system (cf. Section 5.3.1). This latency constraint thereby scales with the acceler-

ation factor of the neuromorphic computing system with respect to the biological timescale, which is usually at the order of ms. The transmission jitter, i.e. the variation in transmission latency, has to be kept as low as possible, as timing variations introduced by the technical network implementation would directly influence e.g. STDP learning at the receiving emulated synapses (cf. Section 5.3.2), as well as integration of spikes from different sources.

Both, the latency and jitter constraint can be fulfilled by delaying received spike events at the destination node by a fixed amount of time which is defined by the modelled axonal delay and must be larger than the maximum possible transmission latency through the network, including the processing time at the source and destination (cf. Section 5.4 and Section 5.6). At the destination, events generally have to be sorted if the delays between a source- and destination neuromorphic node are inhomogeneous, or a destination node receives event streams from multiple source nodes.

In order to optimise the throughput in a packet-based network like EXTOLL with relatively large header sizes, as compared to specialised event communication networks, this theses introduces the concept of so-called *bucket buffers*. These accumulate spike events to form larger network packets, targeting common destination nodes. Thereby, the ratio of the header size with respect to the payload can be optimised. According to the used packet types which are summarised in Appendix B.2, the payload efficiency of the presented implementation using the EXTOLL network ranges between 20 % and 94 %.

Chapter 6 starts by formally defining properties of an accumulation bucket. For this purpose, three criteria are defined for when a packet has to be closed and sent across the network while accumulating events for a common destination (Section 6.1.1), namely when it is full with respect to the network's MTU, when one of the accumulated events crosses a timeout threshold with respect to its arrival timestamp, as well as if the bucket is required for the accumulation of events towards another destination. The latter case can only occur, if there are more appearing destinations than bucket buffers available.

Furthermore, different strategies are defined for assigning the different event destinations to the available buckets (Section 6.1.2). These can be either static, meaning that a given destination is always mapped to the same bucket, or dynamic if the assignment is determined based on the current operating state of all buckets.

Following these definitions, a mathematical method, using the concept of Markov Chains, is derived for evaluating the expected number of events that can be accumulated and how long this is expected to take (Section 6.2), if the average event rate is high enough to not exceed the timeout condition first. This analysis is quantitatively carried out for the expected accumulation length with different static assignment strategies (cf. Section 6.1.2.1) and exemplary destination distributions in Section 6.3.

A metric is proposed in Section 6.3.5 to approximate the expected number of accumulatable events, namely the ratio of the most probable destination's probability to the sum of all other destinations' probabilities at the particular bucket (Equation (6.58)). This metric is used in Section 6.3.6 for a proof of concept optimisation of the static assignment mapping of destinations with respect to their probability of occurrence. In general, the assignment to- as well as the number of available buckets should be optimised in a way that very frequent destinations are accumulated in individual buckets.

Other destinations assigned to those same buckets should be significantly less frequent in order to not severely disturb the accumulation of those events with the more probable destination. From the resulting plots from the different prerequisites, one can read the required number of buckets such that buckets can be expected to be filled before a conflict will occur.

The distribution of event destination probabilities itself depends on the activity of the neural populations, as well as their mapping to the multitude of neuromorphic chips. As described in Section 6.1.2.3 and Section 6.3.1, these activity numbers might be a-priori unknown to the user, making it impossible to optimise the probability distribution of destinations by adjusting the mapping of populations. In that case, the assignment of destinations to buckets should be dynamically optimised directly in the communication hardware (the FPGA firmware).

As is argued in Section 6.2.2.5 that the Markov Chain method is not applicable to dynamic assignment strategies (cf. Section 6.1.2.2), a simulation is additionally set up in order to evaluate the performance of the dynamic RoundRobin strategy (AS.3). RoundRobin is representatively evaluated amongst AS.3 to AS.6 on page 79, as it is argued in Section 6.1.2.3 that the other dynamic strategies will in the worst case converge against RoundRobin assignment. Besides that, the simulation is validated against the mathematical approach by simulating the previously analysed static assignment strategies. The obtained stochastic simulation results thereby match the expectations from the Markov Chain approach to the extent of simulation accuracy.

In summary, it can be found that for normally distributed destinations, RoundRobin Assignment on average has the same performance as the Modulo assignment at the worst performing buckets. In any case a uniform destination distribution is the worst possible case, as any accumulating destination will be disturbed equally frequent by any other destination.

The expected time it takes, to accumulate a network packet until conflict has not been specifically analysed. This decision is justifiable by the fact that the accumulation time (as argued in Section 6.2.4.3) linearly depends on the average firing activity of the populations contributing to a bucket's input event stream. Effectively, the accumulation time is thereby linearly connected to the expected accumulation length with an a-priori unknown factor. A quantitative analysis without a justified assumption on the contributing firing activity (which varies with the modelled neural network and training) would not be expressive and is therefore omitted here. However, the requirement that a destination conflict should on average occur after the timeout constraint closes a packet now leads to a required number of buckets via the obtained analysis- and simulation results. Equation (6.54) relates this requirement constraint to the size of the packet header as well as the size of a single event and the desired limit for the header overhead or payload efficiency, respectively. For the EXTOLL packet headers (cf. Appendix B.2) and the BSS-2 event coding scheme (cf. Section 7.3.3), the ratio of header size to event size is $\frac{32\,\mathrm{B}}{8\,\mathrm{B}} = 4$, so to reach a header overhead below a threshold o_h or a payload efficiency above a threshold e_pl, at least

$$\mathbb{E}\left[N_\mathrm{acc}\right] \geq 4 \cdot \frac{1-o_\mathrm{h}}{o_\mathrm{h}} = 4 \cdot \frac{e_\mathrm{pl}}{1-e_\mathrm{pl}} = 4 \cdot \frac{e_\mathrm{pl}}{o_\mathrm{h}} \tag{9.1}$$

events have to accumulated. By choosing values for the desired header overhead or payload efficiency threshold and plugging the result of this equation into one of the result plots in Chapter 6, one arrives at a required minimal configuration of the bucket system under the respective destination

distribution for which that plot has been created. Thereby two things should be noted: First, the EXTOLL network's maximum payload size (MTU) of 496 B, corresponding to 62 to 124 events, depending on whether they are packet as doublets or singlets and second, the required accumulation time still depending on the mean event rate. The latter caveat has to be kept in mind when interpreting the relatively low number of events transported per packet during the Synfire Chain experiment in Section 8.7, where the event rate is quite low.

Chapter 7 in detail presents the event communication architecture which has been implemented during this work. Here, one important contribution is the synchronised interpretation of transmitted timestamps on different BSS-2 FPGAs in the EXTOLL network by use of the global interrupt mechanism, developed by (Burkhardt 2007, 2012).

Events arriving from the BSS-2 neuromorphic ASIC are first indexed into a lookup table memory to determine the assignment to a specific bucket buffer and to provide a translation to the synapse address in the context of the receiving ASIC. Lookup entries are marked invalid by default at FPGA reset. Thereby, events yielding an invalid, i.e. undefined lookup entry are dropped. This can be used to filter out events that shall only be logged in the experiment trace but are not meant for an inter-chip connection.

The buckets are configured to target a specific destination in the EXTOLL network which can also be a multicast group. The accumulation time is limited by two static timeouts, one for the whole packet and one for the ISI at the bucket's input interface. Additionally, the buckets' configuration space contains the respective axonal delay that is added to every event's timestamp and used at the destination to eliminate the network transmission jitter including accumulation. As stated in Equation (5.4), this value has to be large enough to contain all parts of the delay between creation of the event on the source chip until the reception at the destination FPGA. According to (Alexander Schmidt 2017) the transmission latency and -jitter between the destination FPGA and the receiving synapse driver is handled inside the ASIC by adding another small delay to the timestamp and therefore does not have to be included in the timestamp across the network.

If the axonal delay value is configured too small, events will be dropped at the destination FPGA as they will arrive after the globally synchronised systime has passed their timestamp value. This effect is also observed in the Synfire Chain experiment in Section 8.7 (cf. Figure 8.26 and Figure 8.28) where it is used to estimate a value for the network delay. However, the receiving side also allows to deactivate this check by configuration. The implemented delay buffer at the receiving side allows to delay spike events for up to 2^{14} clock cycles for biological timescale delays up to 130 ms with the BSS-2 speedup of 1000, as has been motivated in Section 5.3.1 with reference to (Swadlow et al. 2012).

In Section 7.5.3.1, the network latency between two BSS-2 FPGAs, connected to the same EXTOLL Tourmalet card, has been estimated to approximately 1 µs. This expectation has been confirmed by measurement in Section 8.5 and indirectly also by the Synfire Chain experiment in Section 8.7. This delay value is expected to scale in steps of approximately 75 ns in larger networks for each hop across another Tourmalet node between the source- and destination FPGA, as the EXTOLL network cards operate at a much higher frequency than the BSS-2 FPGAs.

The design of a bridging interface unit between the BSS-2 system configuration- and status readout bus (*Omnibus*) and the EXTOLL Registerfile is described in Section 7.6. This has been necessary to enable configuration and status readout of the EXTOLL Interrupt and Barrier units globally across the network through the existing system access paradigm, mastered by the FPGA-internal *Playback Executor* unit which is programmed via user-defined *Playback Programs*.
Overall, the communication architecture described in this thesis has been designed in a parametrisable way such that e.g. the number of bucket buffers or parallel data paths can be selected at compile time. This design approach and methodology is described in Section 7.8.

Finally, Chapter 8 describes the methods employed for verifying and testing the described implementation (cf. Section 8.1), as well as considerations for the physical FPGA implementation (cf. Section 8.2) and the measures taken to seamlessly integrate the spike event communication into the BSS-2 software stack (Section 8.4).
Last but not least, Sections 8.5 to 8.7 describe the measurements and experiments done on the system for characterising and demonstrating the synchronisation of multiple BSS-2 FPGAs including spike event communication across the EXTOLL network.
To begin with, the individual delays on EXTOLL links between the Tourmalet ASIC and the BSS-2 FPGAs have been measured in both directions respectively, which is described and discussed in Section 8.6.1. The results of these measurements are required to precisely configure the EXTOLL Interrupt units to balance out the transmission of global interrupt messages such that every node in the network asserts an interrupt at the same point in time (cf. Section 4.1.2 of this thesis and Section 3.4 in Burkhardt 2012).
Next, the overall accuracy of the global Interrupt operation has been assessed in a minimal network of one Tourmalet and four FPGA nodes, using an oscilloscope to measure the absolute timing relation of the interrupt signals on every involved FPGA node. These measurements are described and discussed in Section 8.6.3, yielding an overall uncertainty result of

$$u_{\mathrm{int}} \approx \pm 5\,\mathrm{clk} = \pm 40\,\mathrm{ns} \quad . \tag{8.11}$$

To our best knowledge, this is the first quantification of the EXTOLL global interrupt accuracy.
Finally, the operation of a neuromorphic network emulation spanning two BSS-2 ASICs is demonstrated in Section 8.7, implementing the model of a Synfire Chain which describes a mechanism of reliable signal propagation in biological neural networks according to (Kremkow et al. 2010). Thereby, the activity crosses the network multiple times between parts of the chain. Experiments conducted on this model have investigated the effect of tuning the inhibitory weight (Figure 8.25), as well as the axonal delay and timeout configurations at the accumulation bucket. Hereby, the estimated and previously measured network transmission delay values could be confirmed.

In summary, all considerations and developments, carried out in this thesis led to a functional system which can be used for meaningful experiments. However, there are still some aspects that can and should be improved or complemented in future work to increase the usability and application

coverage. The next Section will elaborate in detail on these aspects.

9.2 Outlook and Discussion

With the current implementation of the event communication architecture (as described in Chapter 7), missing a mechanism for merging multiple event streams from different locations, only one event stream from a single source node can be processed on the receiving side (for a description of the underlying problem cf. Section 7.5.2). The algorithmic solution for this problem is commonly referred to as *priority queue*.

In the BSS-1 system, a priority queue implementation based on a binary heap sort algorithm has been implemented (Scholze, Henker, et al. 2010). However, this algorithm has a logarithmic time complexity, because the insertion and retrieval of events in a binary tree data structure was implemented using a single dual-port RAM, sequentially executing the binary search for the correct insertion location. This might be justifiable if spike events sparsely arrive at the sorting interface with occasional bursts. However in the limit case of high event rates using the full bandwidth of the event interface (one event every clock cycle, compare Section 7.1.2) and especially with support for very large delay values as provided with the current implementation, this poses a major bottleneck. Furthermore, large accumulated event packets containing lots of events to be merge-sorted will also lead to an increased required depth of the heap-sorting tree.

Alternative implementations of priority queues with constant insertion and retrieval complexity are compared and discussed in (Kohútka 2022). The probably most promising priority queue architecture with respect to large buffer sizes while providing constant response time as needed for the merging of event streams is the *Heap Queue* (Kohútka et al. 2018). This is basically a pipelined version of the binary heap sort algorithm and uses one dual-port RAM per tree level. The response time for insertion and retrieval of an element is reported to be two clock cycles. Therefore it will be necessary to either drive the merging unit at double clock speed with respect to the remaining event communication architecture, or to interleave the insertion and retrieval access through two individual *Heap Queues*. When using this kind of priority queue, the sorting on the sending side also becomes obsolete (cf. Section 7.3.1). With adding the merging unit, the design will become generally applicable to scale up the BSS-2 system.

Another aspect that is currently not supported is accumulating events towards more destinations, than buckets are available in the implemented design. This is caused by the fact, that the destinations are currently configured directly at the bucket units rather than the destination mapping lookup table. If there are more destinations to be mapped to accumulation buckets and the FPGA resources do not suffice to further increase this design parameter, the bucket units need to be adapted to also allow relabelling in case of a conflict condition at the input (cf. Section 7.3.3 and Chapter 6). In this case, the static bucket assignment in the destination mapping lookup table (cf. Section 7.3.2) can be programmed in a way, that multiple neuron labels point to the same bucket-id. For this to work, the lookup tables would still have to contain the assigned bucket id, but additionally also the network destination which is currently configured at the bucket.

In this case it would be highly beneficial to optimise the bucket assignment using a strategy similar

to the one proposed in Section 6.3.6. Although this optimisation strategy might also be worthy for further optimisation as pointed out there, it serves as a good starting point for assignment optimisation.

However, as pointed out before, it could occur that the abundance distribution of destinations among the events from the user-implemented neural network model on the distributed neuromorphic computing system is not a-priori known to the user. This might e.g. be the case if the model itself is to be investigated using the neuromorphic emulation and the firing rates of the specific populations cannot be trivially estimated in advance. In this case, the static assignment can also not be optimised in advance and a dynamic assignment, which autonomously optimises the performance at run-time will be beneficial.

In order to implement a dynamic bucket assignment, more complex changes to the communication architecture are necessary. On the one hand, the lookup of the assigned bucket has to be separated from the network destination lookup. On the other hand, the bucket assignment tables have to be dynamically updated, based on the current state of all buckets, as defined in Section 6.1.2.2. A possible design architecture for such a dynamic event accumulation system is proposed in Appendix C.

A more simple improvement to the current event communication architecture concerns the dropping of over-delayed events. Currently, these are only dropped after reception and decoding at the destination FPGA. However, if packets wait a long time for receiving a grant to the sending network interface, it could be advantageous to already check for a possible timestamp timeout at the source FPGA, as also done in (Grübl 2007) for spike event communication between Spikey chips. Thereby precious network bandwidth could be saved by not injecting futile payload. However, this check would take place after encoding the events into the packet, as the accumulation buffer is currently placed behind the encoding stage. Therefore, the resulting drop would affect the whole packet of accumulated events and thereby impose a trade-off between saved network bandwidth and the number of dropped events that could still have arrived in time. An alternative would be to accumulate events before encoding them. This would allow to drop individual events that would not be able to arrive at the destination node in time because the network interface request took too long.

Generally, the presented communication architecture is also applicable to interconnect BSS-1 Wafer Modules. However this would also need some adaptations regarding the interjection of the spike event data stream, as in BSS-1 eight HICANN ASICs were connected to one FPGA. Also the overall FPGA design was structured differently (cf. Thommes 2018). With some adaptations in the destination synapse address mapping at the lookup tables both systems could probably even be operated together, which however might also pose practical issues on the modelling side, regarding their different acceleration factors (10^3 in BSS-2 vs 10^4 in BSS-1). Besides that, the software stacks of BSS-1 and BSS-2 would probably need large adjustments to achieve compatibility.

The particular implementation of the presented event communication architecture can be further optimised at various places. One point would be to optimise the UT encoding scheme to not simply pad datagrams if they are smaller than the output width, but rather place them partly in the datagrams

as they fit. Thereby the bandwidth efficiency could be largely improved in the case when the event stream from the chip provides single events with gaps in between such that they cannot be compacted by the module described in Section 7.1.2.

Furthermore, the configuration procedure for the destination-mapping lookup tables can be optimised in a way that both (all) tables are written with one write access to the configuration bus. This however would require a manual hookup to the RAM interface provided by the Registerfile generator (cf. Section 4.2.4 and Section 7.3.2). This is not trivial because it has to be ensured, that regenerating the registerfile will not destroy the adapted interface attachment.

Regarding the design verification and testing simulation interface, the DPI interface which has been described in Section 8.1.4 may be finally implemented and tested with the modified libRMA. However, as already discussed in the respective section, the benefit of this additional simulation over the existing co-simulation is debatable, as the host-communication can already be successfully tested using the real system and the NHTL unit is extensively verified using the extended UVM testbench from (Thommes 2018).

The problem of a too large bandwidth requirement when simultaneously sending spike events to peer FPGAs and trace data to the host at full rate (cf. Section 8.2.2 on page 155) can be solved in different ways. First, one could increase the trace memory buffer size in order to store all trace data until the end of the experiment and only report them to the host software afterwards. Second, one might drop any trace event that cannot be buffered. The latter option is not preferable due to the obvious data loss. The first option, however has been implemented by Robin Heinemann for the existing Ethernet-based design. This involved the application of a Xilinx® IP DMA controller for the DDR3 memory banks, available on the FPGA boards. However, this led to a largely increased recourse requirement for the DMA controller, which probably has to be optimised before it can fit into the design at hand (cf. Section 8.2.1).

Another improvement by a factor of two in both latency and bandwidth can be achieved by trying to increase the operation frequency at the EXTOLL partition. Thereby the throttling of the Tourmalet link towards the FPGAs by that factor would not be necessary anymore. Apart from that, slightly increasing the operation frequency for compatibility with the original Tourmalet frequency of 630 MHz as compared to the current adaptation to 600 MHz would ease the network scaling through the use of standard EXTOLL network cards.

Finally, on the BSS-2 software stack, further integration of the multi-chip experiment flow, described in Section 8.6.4 will be needed for full user-support of multi-chip neuromorphic experiments on BrainScaleS-2. This will probably include the libraries *Quiggeldy* for automatic setup reservation, *Calix* for the global interrupt delay measurement and configuration as well as *Pynn*, *PyTorch* and *Grenade* for the overall experiment description as well as the mapping and connecting of neural networks across several ASICs (cf. Section 3.3). Especially the latter two aspects are of great importance to the high-level usability of the multi-chip BSS-2 system. As indicated in Figure 6.1 and Figure 6.2 in the preface of Chapter 6, the concrete mapping of neural populations to chips determines the distribution of event destinations at the respective source node and thereby also the stress onto the accumulation system. A strategically favourable placement of those populations, thereby

constitutes a first order optimisation of the accumulation process. Generally, populations that are expected to communicate frequently should be placed as near as possible with regard to the overall network topology and in the best case on the same chip. Also, frequent destination populations should not be spread across many destination nodes.
Neural event projections that target multiple destination nodes at the same time can be realised by defining multicast groups in the EXTOLL network. The possibility for mapping spike events coming from the neuromorphic chip to a multicast destination is already implemented in the lookup tables in the current implementation (cf. Section 7.3.2). However, the EXTOLL software libraries, as well as the Extoll Management Program (EMP) do not yet support programming the multicast routing tables on the Tourmalet ASICs. Therefore, this will also be needed in some layer of the BSS-2 software stack to support spreading out activity across multiple destination chips.

Part IV

Appendix

A Mathematics derivations

A.1 Derivation of the Dennard Scaling Law

The Dennard scaling law (Dennard et al. 1974) connects the scaling of the physical size d of a MOSFET transistor to its performance characteristics like its power consumption P and switching frequency f. The switching power P of a digital circuit is proportional to its capacitance C, the frequency at which it is driven and the square of its operation voltage V.

$$P \propto C \cdot f \cdot V^2 \tag{A.1}$$

The capacitance at first, is proportional to the area over distance, so

$$C \propto \frac{A}{d} \propto d \qquad . \tag{A.2}$$

Second, as the electric field is proportional to the voltage over distance, the voltage can be scaled proportional to the transistor size while keeping the electric field constant

$$\begin{aligned} E \propto \frac{V}{d}, \quad E = \text{const} \\ V \propto d \qquad . \end{aligned} \tag{A.3}$$

Lastly, the current I is proportional to the capacitance multiplied by the momentary change of voltage

$$I = C \cdot \dot{V} \qquad . \tag{A.4}$$

Keeping the voltage change rate $\dot{V}$ constant, this means that the current will also scale linearly with the transistor dimension as well as the transition time Δt. As the switching frequency is reciprocally connected to the transition time, it will scale anti-proportional with the size:

$$f \propto \frac{1}{d} \qquad . \tag{A.5}$$

Putting all this together, leads to a quadratic scaling of a circuits power consumption with the scale of its length dimension, i.e. its area:

$$P \propto d \cdot \frac{1}{d} \cdot d^2 = d^2 \propto A \qquad . \tag{A.6}$$

A.2 Poisson Distribution Statistics

A.2.1 Derivation of the Poisson ISI Distribution

The following derivation of the Poisson ISI distribution leans on the description given in (Heeger 2000).

The poisson probability distribution defines the probability to find n spikes during a time interval $\Delta t = t_2 - t_1$ as

$$P\{n \text{ spikes during } \Delta t\} = e^{-\langle n \rangle} \frac{\langle n \rangle^n}{n!} \tag{A.7}$$

where $\langle n \rangle$ is the average spike count in that interval, given by

$$\langle n \rangle = \int_{t_1}^{t_2} r(t) dt \ . \tag{A.8}$$

The distribution of time intervals between two spikes (the Inter Spike Interval (ISI)) can be derived through the probability for a single spike to occur after a time interval τ

$$P\{\text{next spike occurs before } \tau\} = 1 - e^{-\langle n \rangle} \tag{A.9}$$

which can be identified as the cumulative probability distribution for at least one spike in that interval. By calculating the time derivative of (A.9) one arrives at the wanted probability distribution for the ISIs:

$$p(\tau) = \frac{d}{dt}\left(1 - e^{\langle n \rangle}\right) = \frac{d\langle n \rangle}{dt} e^{-\langle n \rangle} \tag{A.10}$$

With a constant spike rate $r(t) = r$ the average spike count becomes $\langle n \rangle = r\Delta t$ and the ISI distribution becomes

$$p(\tau) = re^{-r\tau} \quad . \tag{2.4}$$

A.2.2 Adding Poisson Distributions

Here a proof is provided that the sum two independent Poisson distributions again yields a Poisson distribution.

Let the random variable X be poisson distributed with rate μ and Y be poisson distributed with rate λ:

$$\begin{aligned} \mathscr{P}(X = m) &= \frac{\lambda^m}{m!} \cdot e^{-\lambda} \\ \mathscr{P}(Y = n) &= \frac{\mu^n}{n!} \cdot e^{-\mu} \end{aligned} \tag{A.11}$$

then

$$
\begin{aligned}
\mathscr{P}(X+Y=k) &= \sum_{i=0}^{k} \mathscr{P}(X+Y=k, X=i) \\
&= \sum_{i=0}^{k} \mathscr{P}(Y=k-i, X=i) \\
&= \sum_{i=0}^{k} \mathscr{P}(Y=k-i) \cdot \mathscr{P}(X=i) \\
&= \sum_{i=0}^{k} e^{-\mu} \frac{\mu^{k-i}}{(k-i)!} \cdot e^{-\lambda} \frac{\lambda^{i}}{i!} \\
&= e^{-(\mu+\lambda)} \frac{1}{k!} \sum_{i=0}^{k} \frac{k!}{i!\,(k-i)!} \mu^{k-i} \lambda^{i} \\
&= e^{-(\mu+\lambda)} \frac{1}{k!} \sum_{i=0}^{k} \binom{k}{i} \mu^{k-i} \lambda^{i} \\
&= \frac{(\mu+\lambda)^{k}}{k!} \cdot e^{-(\mu+\lambda)} \\
&= \frac{\delta^{k}}{k!} \cdot e^{-\delta}
\end{aligned}
\tag{A.12}
$$

which is a Poisson distribution with rate $(\delta = \mu + \lambda)$.

A.3 Proving total probability in the Markov Transition Matrix

Here proofs are provided that the rows in the Matrix

$$
P_{i,j} = \begin{pmatrix}
P_{\text{other},0} & P_{\text{acc},0} & 0 & 0 \\
0 & P_{\text{acc}}^{d^\star} & P_{\text{other}} & P_{\text{relab}}^{d^\star} \\
0 & P_{\text{acc}}^{d^\star} & P_{\text{other}} & P_{\text{relab}}^{d^\star} \\
0 & 0 & 0 & 1
\end{pmatrix}
\tag{6.15}
$$

sum up to 1 with the definitions given in Section 6.2.2.

A.3.1 Row 1

$$
\begin{aligned}
P_{\text{other},0} + P_{\text{acc},0} &= \sum_{d} P(d) \cdot (1 - P(d \rightarrow b)) \\
&\quad + \sum_{d} P(d) \cdot P(d \rightarrow b) \\
&= \sum_{d} P(d) - \sum_{d} P(d) \cdot P(d \rightarrow b) \\
&\quad + \sum_{d} P(d) \cdot P(d \rightarrow b) \\
&= \sum_{d} P(d) \\
&= 1
\end{aligned}
\tag{A.13}
$$

A.3.2 Rows 2 and 3

$$\begin{aligned}
P_{\text{acc}}^{d^\star} + P_{\text{other}} + P_{\text{relab}}^{d^\star} &= P(d^\star) \cdot P(d^\star \to b) + \sum_d P(d) \cdot (1 - P(d \to b)) \\
&\qquad + \sum_{d \neq d^\star} P(d) \cdot P(d \to b) \\
&= \sum_d P(d) - \sum_d P(d) \cdot P(d \to b) \\
&\qquad + \sum_d P(d) \cdot P(d \to b) \\
&= \sum_d P(d) \\
&= 1
\end{aligned} \tag{A.14}$$

A.3.3 Row 1 with Rate Probability

$$\begin{aligned}
P_{\text{other},0}^\star + P_{\text{acc},0}^\star &= (1 - P_{\text{rate}}) + P_{\text{rate}} \cdot P_{\text{other},0} \\
&\qquad + P_{\text{rate}} \cdot P_{\text{acc},0} \\
&= 1 - P_{\text{rate}} + P_{\text{rate}} \cdot (P_{\text{other},0} + P_{\text{acc},0}) \\
&= 1 - P_{\text{rate}} + P_{\text{rate}} \\
&= 1
\end{aligned} \tag{A.15}$$

A.3.4 Row 2 and 3 with Rate Probability

$$\begin{aligned}
P_{\text{acc}}^{\star\, d^\star} + P_{\text{other}}^\star + P_{\text{relab}}^{\star\, d^\star} &= P_{\text{rate}} \cdot P_{\text{acc}}^{d^\star} \\
&\qquad + (1 - P_{\text{rate}}) + P_{\text{rate}} \cdot P_{\text{other},0} \\
&\qquad\qquad + P_{\text{rate}} \cdot P_{\text{relab}}^{d^\star} \\
&= 1 - P_{\text{rate}} \\
&\qquad + P_{\text{rate}} \cdot \left[P_{\text{acc}}^{d^\star} + P_{\text{other}} + P_{\text{relab}}^{d^\star} \right] \\
&= 1 - P_{\text{rate}} + P_{\text{rate}} \\
&= 1
\end{aligned} \tag{A.16}$$

B Implementation Details

B.1 Used Signal Interfaces

This Section introduces the interface-types, used within the *Event Switch* unit, described in Section 7.1.

B.1.1 Valid-Next Interfaces

```
interface valid_next_if #(
  parameter PORTS = 1,
  parameter WIDTH = 8,
  parameter type TYPE = logic [WIDTH-1:0]
)();
  TYPE  [PORTS-1:0] data;
  logic [PORTS-1:0] valid;
  logic             next;

  modport master (
    output valid, data,
    input next
  );
  modport slave (
    input valid, data,
    output next
  );
endinterface
```

Listing B.1: SystemVerilog definition of a valid-next interface.

The *valid-next* interface is the most generic blocking interface and its SystemVerilog definition is shown in Listing B.1. Data of a certain `WIDTH` or a specific `TYPE`, e.g. a **`packed struct`** is transported from the interface `master` to the `slave`. The interface `master` signals validity of the data to the `slave` by asserting the respective `valid` signal `HIGH`. The valid data is held at the interface until the `slave` signals reception by asserting the `next` signal `HIGH`.
The definition above has a special **`parameter`** `PORTS`, specifying the number of parallel data buses to be signalled with individual `valid` signals, but controlled by a single `next` signal. For any parametrisation different from the default, i.e. `PORTS != 1` this is therefore called a *multi-valid-single-next* interface.

B.1.2 UT Interfaces

The UT interfaces were originally introduced by (Karasenko 2020). Listing B.2a shows the definition of the blocking UT interface. It basically resembles a valid-next interface, but adds an up

```
interface ut_b_if #(
  parameter WIDTH,
  parameter type T = logic
   ↪  [WIDTH-1:0]
)();
  logic valid, next;
  T data;
  int unsigned idx;

  modport master (
    output valid, data, idx,
    input next
  );
  modport slave (
    input valid, data, idx,
    output next
  );
endinterface
```

(a) The blocking UT interface

```
interface ut_b_if #(
  parameter WIDTH,
  parameter type T = logic
   ↪  [WIDTH-1:0]
)();
  logic valid;
  T data;
  int unsigned idx;

  modport master (
    output valid, data, idx
  );
  modport slave (
    input valid, data, idx
  );
endinterface
```

(b) The non-blocking UT interface

Listing B.2: SystemVerilog definition of the UT interfaces.

to 32 bit wide type identifier (`int unsigned idx`). Thereby the signalling of sum-types, as introduced in (Karasenko 2020) is possible and multiple independent data queues can be multiplexed over a single interface.

The non-blocking version, shown in Listing B.2b omits the `next` signal and presents any valid data-index pair only for a single clock cycle. If the `slave` does not read the presented data during the single cycle, e.g. because of a blocking pipeline stage, the data will be lost.

B.1.3 FIFO Interfaces

```
interface fifo_write_if #(
  parameter WIDTH,
  parameter type T =
   ↪  logic[WIDTH-1:0]
)();
  logic shift_in, full, a_full;
  T data_in;

  modport fifo (
    input shift_in, data_in,
    output full, a_full
  );
  modport client (
    output shift_in, data_in,
    input full, a_full
  );
endinterface
```

(a) The FIFO write-interface

```
interface fifo_read_if #(
  parameter WIDTH,
  parameter type T =
   ↪  logic[WIDTH-1:0]
)();
  logic shift_out, empty, a_empty;
  T data_out;

  modport fifo (
    input shift_out,
    output data_out, empty, a_empty
  );
  modport client (
    output shift_out,
    input data_out, empty, a_empty
  );
endinterface
```

(b) The FIFO read-interface

Listing B.3: SystemVerilog definition of the FIFO interfaces.

For reading from and writing to FIFO queues, there are two distinct interfaces, which are defined in

```
module conv_vn_to_fifo (
  valid_next_if.slave from_vn,
  fifo_write_if.client to_fifo
);
  logic move_data;
  assign move_data = from_vn.valid
   ↪   && !to_fifo.full;

  assign to_fifo.data_in =
   ↪   from_vn.data;
  assign to_fifo.shift_in =
   ↪   move_data;
  assign from_vn.next = move_data;
endmodule
```

(a) Conversion from a Valid-Next to a FIFO interface.

```
module conv_fifo_to_vn (
  fifo_read_if.client from_fifo,
  valid_next_if.master to_vn
);
  logic move_data;
  assign move_data =
   ↪   !from_fifo.empty &&
   ↪   to_vn.next;

  assign to_vn.data =
   ↪   from_fifo.data_out;
  assign to_vn.valid =
   ↪   !from_fifo.empty;
  assign from_fifo.shift_out =
   ↪   move_data;
endmodule
```

(b) Conversion from a FIFO to a Valid-Next interface.

Listing B.3. These interfaces work a little different from the valid-next interfaces, discussed above.

The `client` signals valid input data to the write-interface (Listing B.3a) of the `fifo` by asserting the `shift_in` signal `HIGH`. The `fifo` will always accept this data within one clock cycle, except when it is `full`, which it signals back to the `client`. Shifting data into a `full` `fifo` is considered an error and may lead to undefined behaviour and not only the loss of the current data, but also the corruption of already buffered data. An additional `a_full` signal therefore warns the `client` at a configurable threshold, when the FIFO is almost full.

At the read-interface (Listing B.3b), the `fifo` signals the availability of data to the `client` by de-asserting the `empty` signal `LOW`. The `client` may then request reading that data by asserting `shift_out` `HIGH`. The `fifo` will then present new data if available after one clock cycle or assert `empty` `HIGH` otherwise. The additional `a_empty` signal again warns the `client` at a configurable threshold, when the FIFO is almost empty.

B.1.4 Conversion between Valid-Next and FIFO Interfaces

Although, valid-next interfaces (Appendix B.1.1) and FIFO interfaces are quite different in their definition, they can however, be easily converted between each other. This conversion can be done by the assignments given in Appendix B.1.4, assuming equal data type (i.e. width) at both interfaces.

The conversion UT interfaces is similar, as the data bus at the FIFO interface now represents the concatenation of both the `data` and `idx` signal buses from the UT interface. The non-blocking version of UT- or general Valid-Next interfaces can also be converted similarly. However, in this case it should be noted that any `full` of the FIFO-write interface will inevitably lead to loss of valid data, as it cannot be stalled. This condition should be reported through a drop-counter with the condition

```
logic drop;
assign drop = from_vn.valid && to_fifo.full;
```
(B.1)

On the other side, the read-interface conversion will now always immediately take available data from the FIFO:

```
assign move_data = !from_fifo.empty;
```
(B.2)

B.2 Used Network Packet Types

This Section will list the EXTOLL RMA (cf. Section 4.2.1) packet formats, used in the NHTL transport layer implementation (cf. Section 7.4). This not only includes communication between host computer and FPGAs but also packet types used for inter-FPGA communication. Most details of the NHTL communication protocol have already been presented in (Thommes 2018). Nonetheless they are shortly repeated here to the reader for reference; for more details the reader may refer to the original thesis of (Thommes 2018). Additionally, packet types and communication modes, specially used for global configuration mastered by the FPGA's *Playback Executor* (cf. Section 3.2.2 and Section 7.6.4) and for spike communication (cf. Section 7.3 and Section 7.5) will be introduced here.

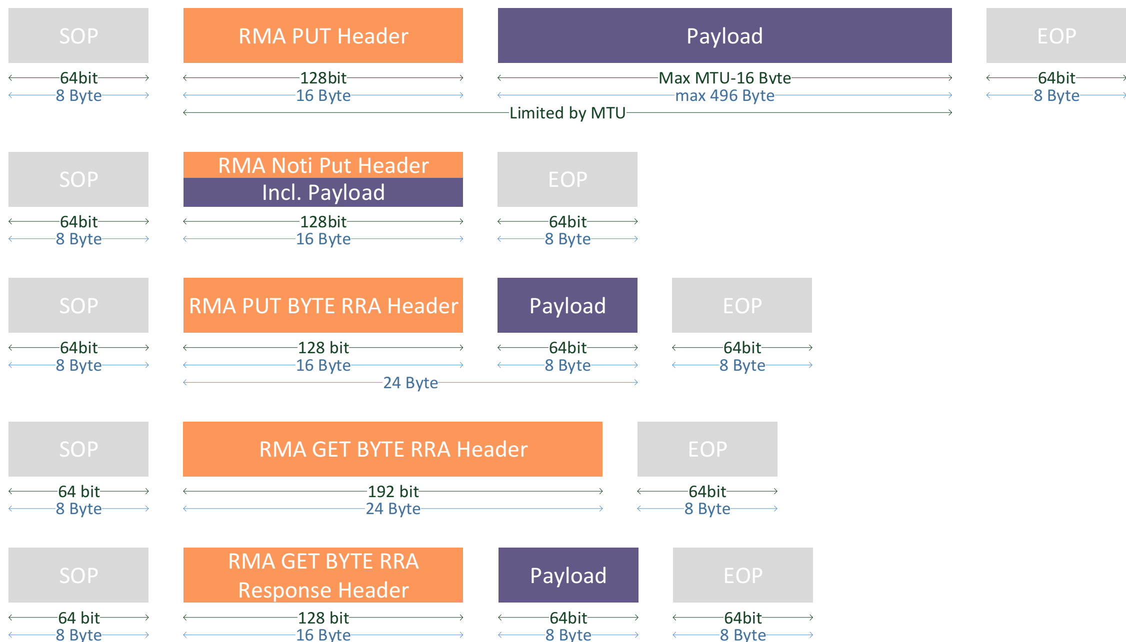


Figure B.1: Overview on the different RMA packet types used in the scope of this thesis. Every packet is preceded by an Start Of Packet (SOP) header (grey), followed by a type-specific RMA header (orange) and different amounts of payload (blue). The MTU for a single packet is 512 B excluding the SOP header, but including the RMA header. Additionally, the EXTOLL NP automatically appends an EOP footer, containing a CRC for the complete packet.

An overview of all used RMA packet types is shown in Figure B.1. From the largest possible packet, the maximum payload efficiency (cf. Equation (5.3) and J. Schmitt 2017) can be determined:

$$E_{\mathrm{pl,max}} = \frac{496\,\mathrm{B}}{496\,\mathrm{B} + 16\,\mathrm{B} + 2 \cdot 8\,\mathrm{B}} \approx 93.9\,\% \tag{B.3}$$

The minimum protocol efficiency can be determined analogueously:

$$E_{\text{pl,min}} = \frac{8\,\text{B}}{8\,\text{B} + 16\,\text{B} + 2 \cdot 8\,\text{B}} = 20\,\% \tag{B.4}$$

RMA PUT packets are used to send host-communication payload in both directions from the *host software* to the *Playback Executor* and from the *Trace Memory* to a ringbuffer region in the hosts main memory.

Notifications are sent in order to used by the *NHTL unit* to inform the *host software* about the amount of trace information sent to the ringbuffer in its memory. The other way round, the *host-software* uses notifications to inform the *NHTL unit* about free space in the ringbuffer. Generally, notifications can be requested as automatic response to any other RMA packet.

Remote Registerfile Accesss (RRAs) are generally communicated using *RMA PUT* and *GET* commands with a modifier bit set. For this, the NHTL restricts itself to the byte-granular access packets (*PUT*- and *GET BYTE*). *GET* requests are generally responded with *GET Response* packets.

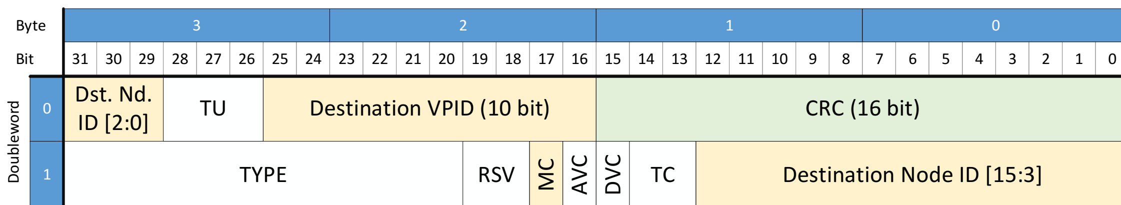


Figure B.2: The SOP Header contains essential routing information like the destination node-id and Virtual Process ID (VPID), as well as a modifier bit for MC messaging. Other modifier bits select the virtual channel (AVC and DVC) and Traffic Class (TC) to be used for the respective packet. The TU field selects whether the packet is destined to the RMA *Completer* or *Responder* unit.

The SOP header format is shown in Figure B.2, containing essential routing information. A packet can be routed using one of two Deterministic Virtual Channels (DVCs) or using an Adaptive Virtual Channel (AVC). The EXTOLL network supports four different traffic classes (separate routes) and up to 2^{16} destinations or 64 multicast groups (if the MC bit is set). While *PUT* packets will always target the *RMA Completer* unit, *GET* packets will always target the *RMA Responder* unit.

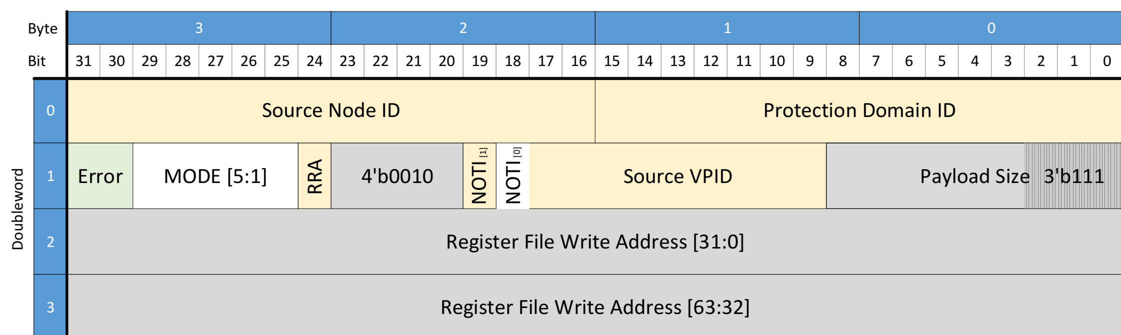
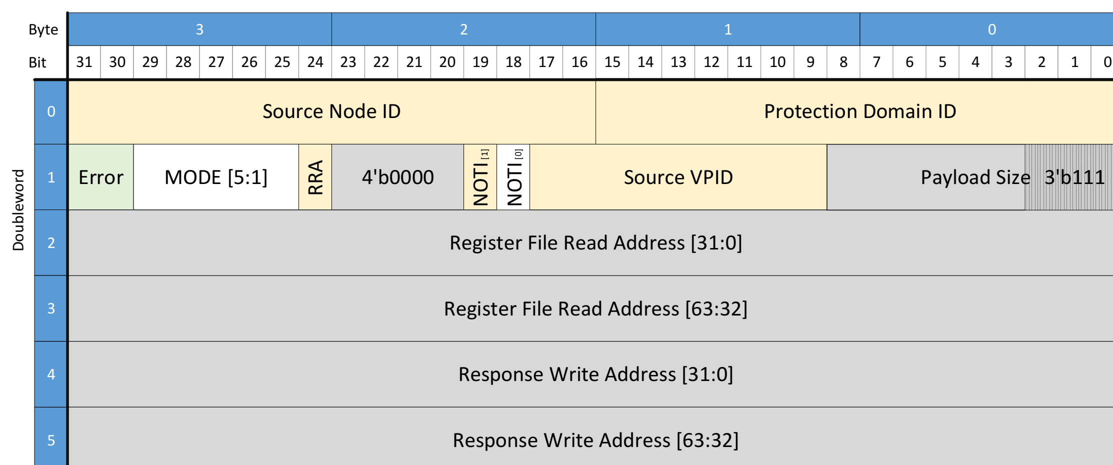
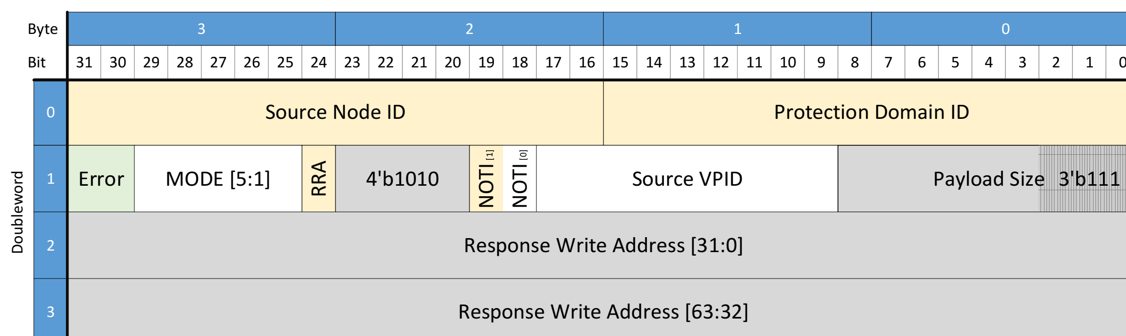


Figure B.3: A *Registerfile-PUT* packet's header, besides the general packet header information, contains a Registerfile address to be accessed in write-mode. The payload size is fixed to one QW (the addressable unit in the Registerfile) and the RRA modifier is set.

In contrast to the SOP header, specific command headers contain information that is required to interpret the packet content at its destination. This generally includes information about its source node as well as a type-specifier, error- and mode-flags, information about the payload size (actually transported as decremented by one for optimised bit efficiency) and the address information from where to read and where to write the accessed payload information. An acknowledgement notification can be requested on either side of the transfer back to the software process issuing the request.



(a)



(b)

Figure B.4: **(a)** A *Registerfile-GET* request packet's header, compared to a corresponding *Registerfile-PUT* request (cf. Figure B.3), carries an additional address field containing the location where the response payload shall be written to. Again the payload size is fixed to one QW and the RRA modifier is set. **(b)** A *Registerfile-GET-Response* packet's header mirrors the source information from the respective *GET* request's (cf. Figure B.4a) source to its own destination. The single address field now carries the response address that has been received from the *GET* request. Again the payload size is fixed to one QW and the RRA modifier is set.

Originally, Remote Registerfile Access where only supported by the NHTL as mastered by the host software. In the context of this theses, support was added for FPGA-mastered Remote Registerfile Access to arbitrary nodes in the network, driven by Omnibus access instructions in the playback program to a special part of the Omnibus address space (cf. Section 7.6.4). This part of the Omnibus address space has to be large enough to contain the full Registerfile address space

at any node in the network. As the Registerfile of the Tourmalet ASIC is larger than the one in the BSS-2 FPGA, this is the limiting factor. The Tourmalet Registerfile has a byte-granular address size of 26 bit. However, as the Registerfile is addressed in units of 1 QW and the three least significant address bits are always clamped to zero, 23 bit are sufficient to describe the address space. Because of the data-width conversion from the 32 bit Omnibus to the 64 bit registerfile (cf. Section 7.6.2), a 24'th address bit is required to map the registerfile address space into the Omnibus.

The RRA packet header formats are shown in Figure B.3, for *PUT* requests and Figure B.4 for *GET* requests and *GET-Response* packets respectively. The mastering node will send a *PUT* or *GET* request packet to the remote node and receive a *GET-Response* respectively. In case of an invalid address request, the *GET-Response* will have an Error-flag asserted. *PUT* requests do not produce a response.

The NHTL protocol sends playback programs directly to a FIFO queue in the FPGA unit and returns trace data to a ringbuffer memory region at the host. These data transfers are done using *RMA PUT-QW* packets. Notably, a full playback buffer will stall the reception of packets at the FPGA and thereby may eventually stall the RMA requester unit at the remote Tourmalet ASIC. In the opposite direction, a full ringbuffer will stall the insertion of packets into the network and thereby also make the trace buffer fill up, eventually stalling the trace path of the *Playback Executor* unit. Both cases have been observed during system tests, the latter playing a role in the effects, described in Section 8.4.7.

Live spike event data are also sent using the same packet type coded as stream of UT datagrams (cf. Section 7.3 and Section 7.4), but not to the host, but directly to other FPGAs where they are received and further processed for timed execution (cf. Section 7.5).

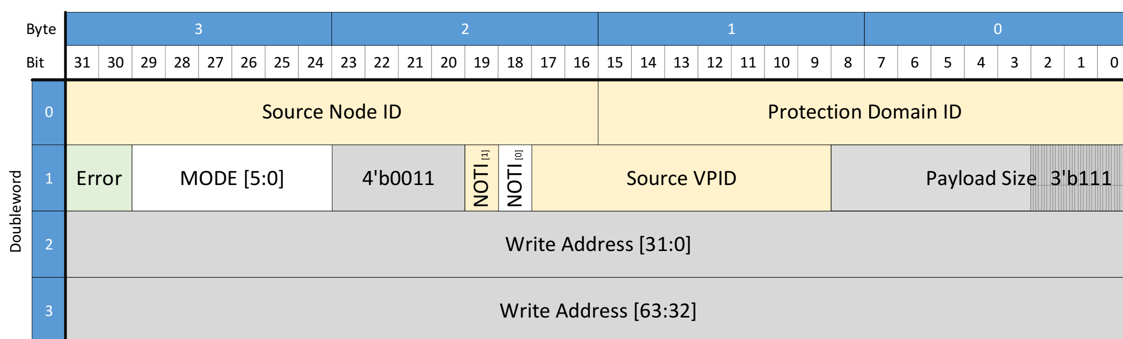
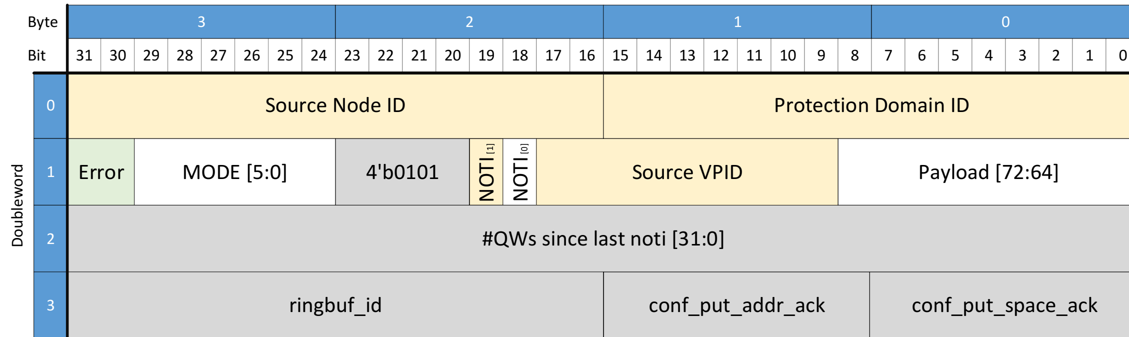


Figure B.5: A normal *RMA-PUT* request can carry up to 496 B of payload data to subsequent addresses in destination node's memory-space. The Translate Enable (TE) bit determines whether the accessed address is virtual and fist has to be translated by the EXTOLL Address-Translation-Unit (ATU).

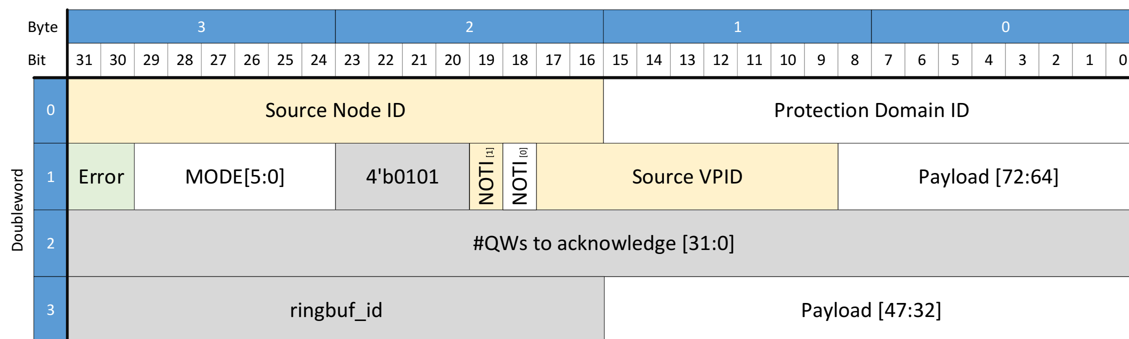
A flow control mechanism for the NHTL transport layer is implemented using notification messages transporting information about the amount of moved data and freed buffer space to the host's ring-buffer memory region. These notification messages are transported using *RMA Notification* packets, shown in Figure B.6.

For a future improvement, a similar notification flow control mechanism could also be implemented in the direction from the host to the FPGA to prevent stalling the network when the playback buffer runs full. Alternatively packets might be accepted and dropped if they cannot be buffered. However

this would break the high-level experiment flow and would also have to be notified back to the host-software to throw an exception and abort the user's experiment program.



(a)



(b)

Figure B.6: *RMA Notification* packets are used for communicating protocol information in both directions between the host software and the FPGA's NHTL unit. **(a)** In the direction from the FPGA to the host, the notification message contains the number of payload QWs sent to the host ringbuffer since the last notification, labelled by an id-number of the particular ringbuffer. Besides that, the notification contains some debug information, namely the number of configuration changes regarding the NHTL unit's ringbuffer controller since the last notification. **(b)** In the opposite direction, the host software notifies the NHTL unit about the number of QWs read from the ringbuffer since the last notification and thereby the amount of freed space, again labelled by an identifier regarding the particular ringbuffer instance.

C Dynamic Bucket Concept

In Chapter 6, the theoretical properties and limitations of the event accumulation process have been studied under different input distribution constraints and assignment strategies. As has been argued in Section 9.1 on page 190 and Section 9.2 on page 195, it might not always be possible for a user or mapping algorithm to optimise a static strategy for assigning event destinations to particular buckets. Therefore it has been proposed that dynamic assignment at runtime, based on the current state of all buckets, might be advantageous.
A possible implementation concept for dynamic bucket assignment has been developed at the early time of this thesis' work. However, this has not yet been implemented in favour of the more simple design described in Chapter 7. This dynamic assignment design concept will be concisely described in this Chapter. It has been presented at the 8th Annual Neuro-Inspired Computational Elements (NICE) online workshop in 2021 (cf. Thommes, N. Buwen, et al. 2021).

C.1 Overview

An overview block diagram of the proposed dynamic assignment architecture is presented in Figure C.1. Events, arriving from the neuromorphic ASIC are in a first step indexed into a lookup table to retrieve the mapped network- and synapse destination address. In a second step, the network destination is now indexed into another lookup table to retrieve the currently assigned bucket id. This is in contrast to the implemented architecture described in Section 7.3, where only one lookup directly provides the bucket id and the network destination is configured to the bucket units themselves.
If the second lookup yields that currently no bucket is assigned to the given destination, an empty bucket is provided by the *Free Bucket Arbiter* instead, if available. If also no empty bucket is available, an already occupied bucket has to be interrupted in its accumulation process with a conflict condition. In this case the *Occupied Bucket Arbiter* selects a bucket for interruption. This selection can follow one of the strategies, listed and discussed under Section 6.1.2.2 and Section 6.1.2.3 respectively. In terms of critical path length, this arbitration might become problematic with increasing number of buckets, in case the selection is based on an optimisation like e.g. "the bucket containing the most accumulated events" (AS.5) or "the bucket with nearest timeout" (AS.6). In case of *local* arbitration strategies like *RoundRobin* (AS.3) or *Random* assignment (AS.4), the implementation is rather not critical with respect to the timing path length.
That bucket, which is selected and receives the conflicting event must then update all the *Bucket-LUTs* at the parallel input data paths with its new assignment. When a bucket has closed its packet, it requests access to the network interface. This arbitration can again follow one of the said arbitration strategies.
Regarding memory resource requirements, a naive lookup table for the mapping of network destinations to bucket ids is even more challenging than the event destination lookup. While the latter

Figure C.1: Schematic block diagram of the proposed dynamic bucket assignment architecture. In this example, events arrive from two parallel data paths after being mapped to network destinations. The assigned bucket for the mapped destination is acquired from a lookup table. If no bucket is assigned yet, a free bucket will be selected by by the *Free Bucket Arbiter* (on the left). If no bucket is left free (conflict condition), the *Occupied Bucket Arbiter* (on the right) selects a bucket for re-assignment. The assigned bucket is requested to take the event and if granted, the event is stored in its the buffer. When a bucket is full or requested for re-assignment, it requests access to the network interface before sending the accumulated packet away.

has an address size of 14 bit for the BSS-2 event labels, the former would even be eight times larger with the 16 bit EXTOLL node id plus the 1 bit multicast flag. To reduce the memory requirement drastically, the second stage index, retrieved from the first stage may be changed to a shorter Global Unique Identifier (GUID) which is unique with respect to all used node ids in the network. This is possible, as only a small fraction of the theoretically supported 2^{16} EXTOLL nodes will be needed for a large scale BSS-2 system in the near future. The actual 17 bit network destination address will then be retrieved from another lookup table. However this one is sufficient to be implemented only

once for the whole design and may be accessed via an arbiter by the buckets while accumulating events and before sending out the packets.

As the *free bucket arbiter* and the *occupied bucket arbiter* can make their decision speculatively before it is needed by a requesting input buffer, they do not cause additional latency. However, as multiple input buffers might request an empty or occupied bucket at the same time, the arbiter implementation must be capable of also giving out multiple grants at the same time if the parallel data paths shall not be serialised by their requests.

C.2 Arbitration Request Pipeline

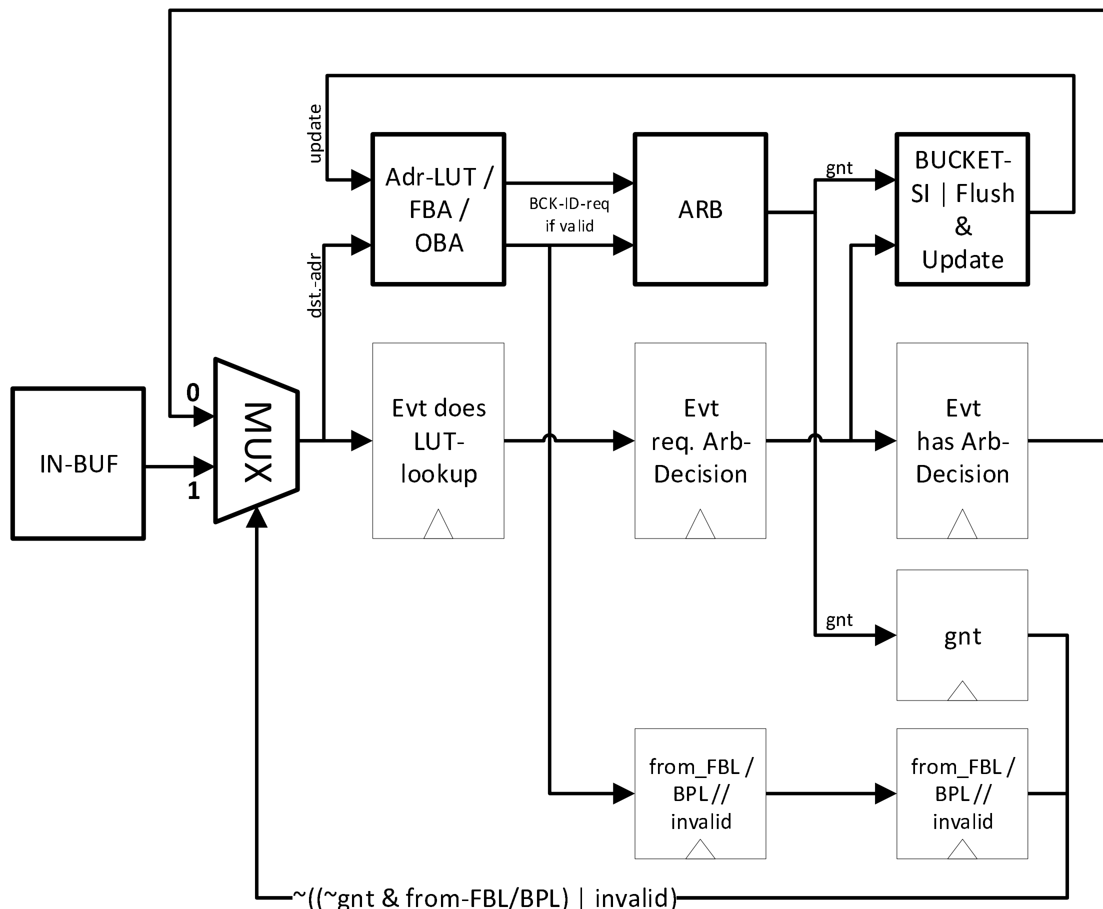


Figure C.2: Schematic block diagram of the bucket request pipeline, symbolised by the *one-hot request* in Figure C.1. Incoming events have to wait a first clock cycle for the result of the bucket lookup. On the second stage, they wait for a grant from the respectively requested bucket. If the bucket is interrupted by a new destination it updates the lookup tables in all the input channels. Events that have already acquired a bucket id from this now out-of-date lookup table need to be invalidated in order to repeat their lookup. During the update, the current event is kept in a third pipeline stage.

In order to ensure full event throughput with one event at every clock cycle, the input data paths have to be pipelined in three stages. A schematic block diagram of this pipeline is shown in Figure C.2. The first clock cycle is spent on retrieving the respective bucket id from the lookup table, followed

by the second cycle, which is spent on requesting the respective bucket to grant this input channel. If granted, the event is stored in the bucket. Otherwise, the pipeline is stalled until the event is eventually taken. However, if the bucket is interrupted by a new destination, it has to update the lookup tables and all waiting events, that are now holding a possibly invalid lookup result, have to be cancelled and repeat their lookup by cycling back to the pipeline start. The third pipeline stage ensures, that the repeated lookup happens after the update has been done. The lookup also has to be repeated, if an empty bucket has not immediately granted the requesting event, because then it probably has been labelled otherwise by another input channel.
Care must be taken to prevent events from endlessly cycling the pipeline as their lookup might be repeatedly invalidated by updates to the lookup table. A possible safety measure would be to implement a cycle counter, that is propagated with the event around the pipeline and which will cause the event to be dropped after a certain number of cycles. This would be similar to the Time To Live (TTL) counter in the Internet Protocol.

C.3 Bucket Finite-State Machine

The behaviour of a dynamically assigned bucket can be described by a Finite-State Machine (FSM) as depicted in Appendix C.3. As this FSM looks rather complicated at the first glance, it shall not be described in full detail here. Rather the following description shall provide guidance instructions of how to read the diagram while giving an overview on the basic function of the bucket.

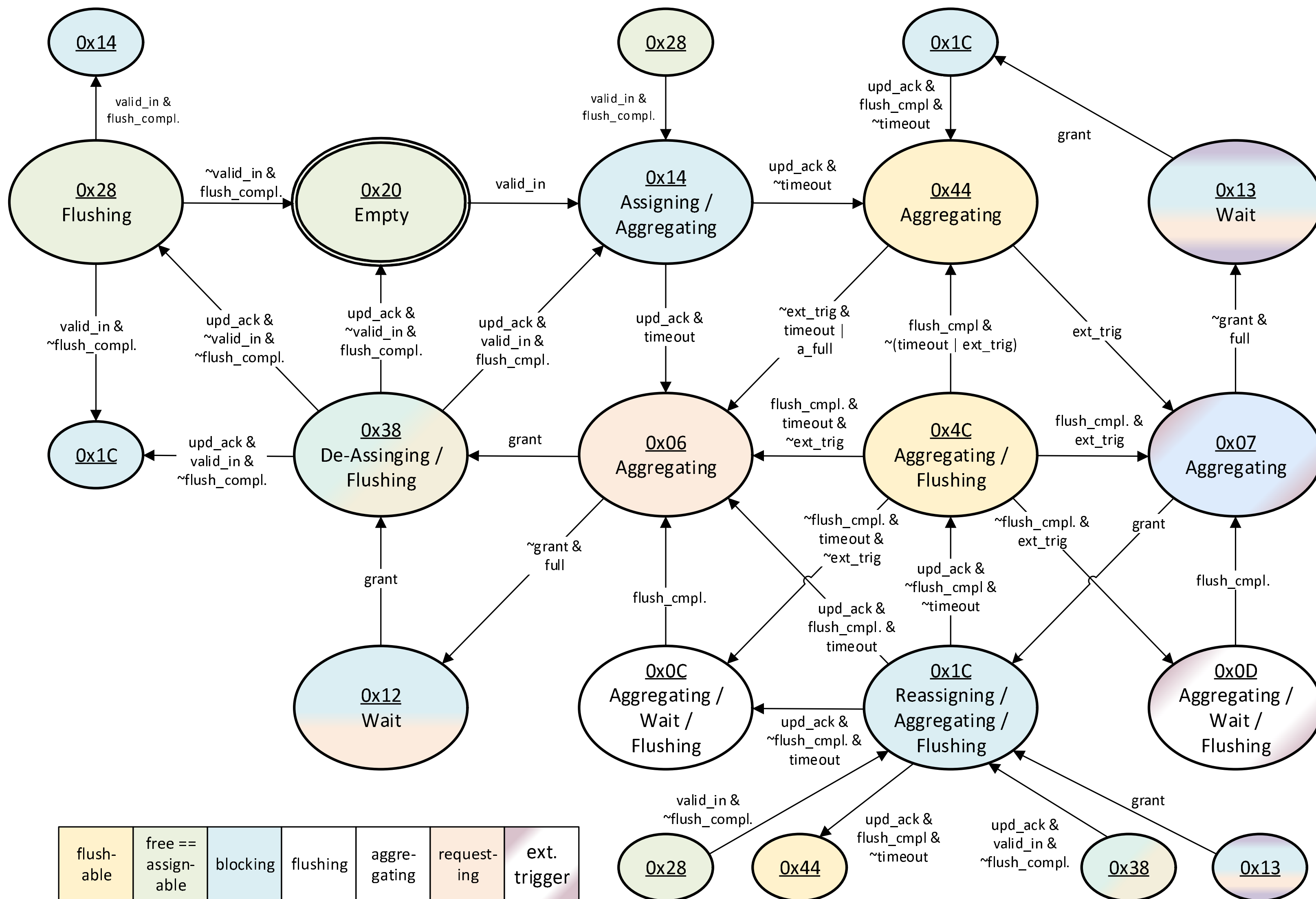


Figure C.3: State diagram of the Finite-State Machine describing the behaviour of a dynamically assigned bucket. States are coded by seven binary state variables that can be partially asserted at the same time. These variables are listed in the legend on the lower left edge and colour coded to the respective states. Hexadecimal numbers on each state also give the exact state code. The initial state is *Empty (0x20)*. Long-range transitions across the diagram edges are depicted by small state-duplicates, i.e. connected by reference.

Generally, the bucket's state is described by seven state variables which have the following meanings:

- **Free:** A bucket is *free*, i.e. empty if it contains no events. In this case it is available for assignment and requests the *Free Bucket Arbiter*.
- **Blocking:** When a bucket changes its assignment state, i.e. is assigned to or de-assigned from a specific destination, it temporarily *blocks* events from the input channels. The lookup tables are updated in this state and requesting events in the input pipelines have to repeat their lookup with a new pipeline cycle.
- **Aggregating:** In this state, the bucket performs its main purpose of collecting, i.e. *aggregating* events for the assigned destination.
- **External Trigger:** In this state, the bucket has received a conflicting request, conveyed by the *occupied-bucket arbiter*. This will lead to a re-assignment and lookup table update.
- **Requesting:** In this state, whether because it has accumulated a full packet or the packet timeout exceeded or it has received a conflicting request, the bucket *requests the network interface* in order to send the closed packet.
- **Flushing:** After the bucket has received a grant from the network interface, it will flush out the aggregated packet.
- **Flushable:** In this state, the bucket is available for flushing by external trigger. It requests the *Occupied Bucket Arbiter* and can thereby be requested with a new destination.

These atomic state variables can partly be asserted at the same time. Importantly, a bucket should be able to accumulate events for a new packet while flushing an already finished one to the network. In Appendix C.3, the particular states are coloured with respect to the currently active atomic state variables. The vector of these state bits thereby completely encodes the state space. Each state block is labelled with the hexadecimal value of the state vector as defined in the Figure's legend. The typical state transitions roughly cycle trough a sequence, as defined by the list above. However, there exist several shortcuts looping back or skipping states in this sequence.

D Acronyms

ADC Analog to Digital Converter

AdEx Adaptive Exponential integrate-and-fire model Gerstner and Brette 2009b

AER Address Event Representation

AL Application-Layer

AMD Advanced Micro Devices

ANN Artificial Neural Network

ANNCORE Analog Neural Network Core

API Application Programming Interface

ARM Acorn RISC Machines / Advanced RISC Machines

ARQ Automatic Repeat reQuest

ASIC Application Specific Integrated Circuit

ATOLL Atomic Low Latency

ATU Address-Translation-Unit

AVC Adaptive Virtual Channel

AWK A programming language designed for scanning text files, named after the surnames of its three authors Alfred V. Aho, Peter J. Weinberger und Brian W. Kernighan.Aho et al. 1988

BRAIN Initiative Brain Research Through Advancing Innovative Neurotechnologies® Initiative

BRAM Block-RAM

BSS BrainScaleS

BSS-1 BrainScaleS-1

BSS-2 BrainScaleS-2

CADC Column-parallel ADC

CAG the former *Computer Architecture Group* within ZITI at the University of Heidelberg

CAM Content Addressable Memory

CI Continuous Integration

CMOS Complementary Metal-Oxide-Semiconductor

CNN Convolutional Neural Network

CPU Central Processing Unit

CRC Cyclic Redundancy Checksum

DAC Digital to Analog Converter

DDR Double Data Rate

DLL Delay-Locked-Loop

DMA Direct Memory Access

DPI Direct Programming Interface

DRAM dynamic random access memory

DSP Digital Signal Processor

DUT Design Under Test

DUV Design under Verification

DVC Deterministic Virtual Channel

E Unit for table-Entries

EINC European Institute for Neormorphic Computing

EMP Extoll Management Program

ENIAC Electronic Numerical Integrator and Computer

EOP End Of Packet

EXTOLL Extended Atomic Low Latency (ATOLL)

FCAA fetch-compare-and-add

FF Flip Flop

FIFO First In First Out

FinFET Fin Field Effect Transistor

fisch FPGA Instruction Set Compiler for HICANN

flit flow control unit

Flop Floating Point Operation

FPGA Field Programmable Gate Array

FSM Finite-State Machine

FU Functional Unit

GAAFET Gate All Around Field Effect Transistor

GALS Globally-Asynchronous, Locally-Synchronous

GIL Global Interpreter Lock

GPIO General Purpose Input and Output

GUID Global Unique Identifier

HBP Human Brain Project

HDL Hardware Definition Language

HICANN High Input Count Analog Neural Network

HICANN-X High Input Count Analog Neural Network with HAGEN Extensions

HMF Hybrid Multiscale Facility

HOL Head of Line Blocking

HPE Hewlett Packard Enterprise

HT HyperTransport

HTAX HyperTransport Advanced Crossbar

HTML Hyper Text Markup Language

HToC HyperTransport on-chip Protocol

IC Integrated Circuit

ILA Integrated Logic Analyzer

IP Intellectual Property

ISI Inter Spike Interval

ISO International Standardisation Organisation

ITU International Telecommunication Union

JTAG Joint Test Action Group

L1 Layer 1

L2 Layer 2

LED Light Emitting Diode

libHBP Former name of the NHTL-Extoll API

libRMA User-level API for RMA communication, provided by EXTOLL.

LIF Leaky Integrate-and-Fire

LP Link-Port

LSB Least Significant Bit

LUT Lookup Table

LVDS low voltage differential signaling

MADC Membrane ADC

MC Multicast

MIN Multistage Interconnection Network

MOSFET Metal Oxide Semiconductor Field Effect Transistor

MPI Message Passing Interface

MSB Most Significant Bit

MTU Maximum Transmission Unit

NDID Node ID

NHTL Network HMF Transaction Layer

NHTL-Extoll Neuromorphic Hardware Transaction Layer via Extoll

NIC Network Interface Controller

NICE Neuro-Inspired Computational Elements

NP Network-Port

NRP Neuro-Robotics Platform

OCP Open Core Protocol

OS Operating System

OSI Open Systems Interconnection

PCB Printed Circuit Board

PCIe Peripheral Component Interconnect Express

phit physical digit

PPU Plasticity Processing Unit

PSP postsynaptic potential

PyNN A Python package for simulator-independent specification of neuronal network models.

QoS Quality of Service

QW Quad Word

RAM Random Access Memory

RF Register File

RFG register file generator

RISC Reduced Instruction Set Computer

RMA Remote Memory Access

RRA Remote Registerfile Access

RTL Register Transfer Language

SAF Store-and-Forward Switching

SCB Scoreboard

SIMD Single Instruction Multiple Data

SIMO Single-In-Multiple-Out

SMFU Shared Memory Functional Unit

SNN Spiking Neural Network

SOP Start Of Packet

SpiNNaker Spiking Neural Network Architecture

SRAM static random access memory

STDP spike-timing-dependent plasticity

STP short-term plasticity

systime system time

TC Traffic Class

TCL Tool Command Language

TDM Time Devision Multiplexing

TE Translate Enable

TRADIC TRAnsistor Digital Computer

TSMC Taiwan Semiconductor Manufacturing Company Limited

TTL Time To Live

TU Target Unit

UDP User Datagram Protocol

UMC United Microelectronics Corporation

USB Universal Serial Bus

UT universal translator

UVC Universal Verification Component

UVM Universal Verification Methodology

VC Virtual Channel

VCT Virtual Cut-Through Switching

VELO Virtualised Engine for Low Overhead

VLSI Very Large Scale Integration

VoIP Voice over Internet Protocol

VOQ Virtual Output Queue

VPID Virtual Process ID

VRHD Virtual Ringbuffer Handler

WAR Write After Read

WAW Write After Write

WHS Wormhole Switching

WSI Wafer Scale Integration

XML Extensible Markup Language

XOR Exclusive OR

ZITI Zentrales Institut für Technische Informatik (Institute of Computer Engineering)

Publications

(Thommes, N. Buwen, et al. 2021): Tobias Thommes, Niels Buwen, Andreas Grübl, Eric Müller, Ulrich Brüning, and Johannes Schemmel (2021). "BrainScaleS Large Scale Spike Communication using Extoll". In: *2021 8th Neuro Inspired Computational Elements Workshop (NICE'2020)*. peer-reviewed extended abstract incl. paper presentation. arXiv: 2111.15296 [cs.AR]
The contents of this publication are presented in Section 8.4.1 and in Appendix C.

(Thommes, Bordukat, et al. 2022): Tobias Thommes, Sven Bordukat, Andreas Grübl, Vitali Karasenko, Eric Müller, and Johannes Schemmel (2022). "Demonstrating BrainScaleS-2 Inter-Chip Pulse Communication using EXTOLL". in: *Neuro-inspired Computational Elements Workshop (NICE '22), March 29 – April 1, 2022*. Virtual Event, USA: Association for Computing Machinery, pp. 98–100. ISBN: 9781450395595. DOI: 10.1145/3517343.3517376. arXiv: 2202.12122 [cs.AR]
The contents of this publication are presented in Section 8.4.1 and Section 8.5.

(Thommes, Grübl, et al. 2023): Tobias Thommes, Andreas Grübl, and Johannes Schemmel (2023). "Optimising Spike Throughput and Latency for Spike-Event Accumulation on Packet-Based Interconnection Networks". In: *7th HBP Student Conference on Interdisciplinary Brain Research*. Frontiers Media SA, pp. 258–264
The contents of this publication are presented in Chapter 6.

(Müller, Emmel, et al. 2023): Eric Müller, Arne Emmel, et al. (2023). *The BrainScaleS-2 Neuromorphic Platform — A Report on the Integration and Operation of an Open Science Hardware Platform within EBRAINS*. DOI: 10.5281/zenodo.8375522
This publication is cited in the preface of Chapter 3 and in Section 8.7.

References

Aamir, Syed Ahmed, Paul Müller, Gerd Kiene, Laura Kriener, Yannik Stradmann, Andreas Grübl, Johannes Schemmel, and Karlheinz Meier (2018). "A Mixed-Signal Structured AdEx Neuron for Accelerated Neuromorphic Cores". In: *IEEE Transactions on Biomedical Circuits and Systems* 12.5, pp. 1027–1037. ISSN: 1932-4545. DOI: 10.1109/TBCAS.2018.2848203.

Abbott, Larry F (1999). "Lapicque's introduction of the integrate-and-fire model neuron (1907)". In: *Brain research bulletin* 50.5-6, pp. 303–304. DOI: 10.1016/s0361-9230(99)00161-6.

Abbott, Larry F and Sacha B Nelson (2000). "Synaptic plasticity: taming the beast". In: *Nature Neuroscience* 3, pp. 1178–1183.

Abeles, M. (1991). *Corticonics: Neural Circuits of the Cerebral Cortex*. New York: Cambridge University Press.

Aertsen, A., M. Diesmann, and M. O. Gewaltig (1996). "Propagation of synchronous spiking activity in feedforward neural networks." In: *J Physiol Paris* 90.3-4, pp. 243–247. ISSN: 0928-4257. URL: http://view.ncbi.nlm.nih.gov/pubmed/9116676.

Aho, Alfred V, Brian W Kernighan, and Peter J Weinberger (1988). *The AWK programming language*. Addison-Wesley. ISBN: 978-0-201-07981-4.

Ajima, Yuichiro et al. (2018). "The Tofu Interconnect D". In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 646–654. DOI: 10.1109/CLUSTER.2018.00090.

Allen, Christina and Charles F Stevens (1994). "An evaluation of causes for unreliability of synaptic transmission." In: *Proceedings of the National Academy of Sciences* 91.22, pp. 10380–10383. DOI: 10.1073/pnas.91.22.10380.

Amari, Shun'ichi (1967). "A Theory of Adaptive Pattern Classifiers". In: *IEEE Transactions on Electronic Computers* EC-16.3, pp. 299–307. DOI: 10.1109/PGEC.1967.264666.

Amari, Shun'ichi (1993). "Backpropagation and stochastic gradient descent method". In: *Neurocomputing* 5.4, pp. 185–196. ISSN: 0925-2312. DOI: 10.1016/0925-2312(93)90006-O.

AMD (2023). *Vivado Design Suite User Guide (UG908). Programming and Debugging*. Version v2023.1.

Amdahl, Gene M. (1967). "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. AFIPS '67 (Spring). Association for Computing Machinery, pp. 483–485. ISBN: 9781450378956. DOI: 10.1145/1465482.1465560.

Amunts, Katrin, Hartmut Mohlberg, Sebastian Bludau, and Karl Zilles (2020). "Julich-Brain: A 3D probabilistic atlas of the human brain's cytoarchitecture". In: *Science* 369.6506, pp. 988–992. DOI: 10.1126/science.abb4588.

Arora, Sanjeev, Tom Leighton, and Bruce Maggs (1990). "On-Line Algorithms for Path Selection in a Nonblocking Network". In: *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, pp. 149–158. ISBN: 0897913612. DOI: 10.1145/100216.100232.

Arshak, Khalil, Essa Jafer, and Christian Ibala (2006). “Testing FPGA based digital system using XILINX ChipScope logic analyzer”. In: *2006 29th International Spring Seminar on Electronics Technology*. IEEE, pp. 355–360. DOI: 10.1109/ISSE.2006.365129.

Baddeley, Roland, L. F. Abbott, Michael C. A. Booth, Frank Sengpiel, Toby Freeman, Edward A. Wakeman, and Edmund T. Rolls (1997). “Responses of neurons in primary and inferior temporal visual cortices to natural scenes”. In: *Proceedings of the Royal Society B* 264, pp. 1775–1783.

Ben Abdallah, Abderazek and Khanh N. Dang (2022). *Neuromorphic computing principles and organization*. Cham: Springer. ISBN: 978-3-030-92525-3. DOI: 10.1007/978-3-030-92525-3_1.

Beneš, Václav E. (1965). *Mathematical theory of connecting networks and telephone traffic*. Mathematics in science and engineering ; v. 17 17. New York: Academic Press. ISBN: 978-0-08-095523-0. URL: http://www.sciencedirect.com/science/book/9780120875504.

Bi, Guo-qiang and Mu-ming Poo (1998). “Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type.” In: *Journal of Neuroscience* 18.24, pp. 10464–10472. ISSN: 0270-6474. DOI: 10.1523/JNEUROSCI.18-24-10464.1998. URL: http://www.jneurosci.org/content/18/24/10464.

Billaudelle, Sebastian (2017). “Design and Implementation of a Short Term Plasticity Circuit for a 65 nm Neuromorphic Hardware System”. Master thesis. Ruprecht-Karls-Universität Heidelberg.

Billaudelle, Sebastian (2022). “From transistors to learning systems. circuits and algorithms for brain-inspired computing”. PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Black, Paul E. (2011). *postman’s sort*. URL: https://www.nist.gov/dads/HTML/postmansort.html (visited on 06/23/2023).

Block, H. D. (1962). “The Perceptron: a model for brain functioning”. In: *Reviews of Modern Physics* 34, pp. 123–135.

Boërio, D., J-Y. Hogrel, A. Créange, and J-P. Lefaucheur (2004). “Méthodes et intérêt clinique de la mesure de la période réfractaire nerveuse périphérique chez l’homme”. In: *Neurophysiologie Clinique/Clinical Neurophysiology* 34.6, pp. 279–291. ISSN: 0987-7053. DOI: 10.1016/j.neucli.2004.08.002.

Bohnstingl, Thomas, Franz Scherr, Christian Pehle, Karlheinz Meier, and Wolfgang Maass (2019). “Neuromorphic Hardware Learns to Learn”. In: *Frontiers in Neuroscience* 2019.13, pp. 1–14. ISSN: 1662-4548. DOI: 10.3389/fnins.2019.00483.

Brunel, N. (2000). “Dynamics of sparsely connected networks of excitatory and inhibitory spiking neurons”. In: *Journal of Computational Neuroscience* 8.3, pp. 183–208.

Burkhardt, Niels (2007). “Fast Hardware Barrier Synchronisation for a Reliable Interconnection Network”. Diploma thesis. Mannheim University.

Burkhardt, Niels (2012). “A Hardware Verification Methodology for an Interconnection Network with fast Process Synchronization”. PhD thesis. Mannheim University.

Buwen, Niels Arwed (2019). “Design and Implementation of a Transport Layer for the Extoll Network Interface in the BrainScaleS Neuromorphic Computing Platform”. Master thesis. Ruprecht-Karls-Universität Heidelberg.

Clos, Charles (1953). “A study of non-blocking switching networks”. In: *The Bell System Technical Journal* 32.2, pp. 406–424. DOI: 10.1002/j.1538-7305.1953.tb01433.x.

Computer Architecture Group (2018). *The CAG Registerfile Generator*. URL: https://github.com/unihd-cag/odfi-rfg.

Cormen, Thomas H., Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein (2009). *Introduction to algorithms*. eng. Third edition. Cambridge, Massachusetts ; London, England: MIT Press, xix, 1292 pages. ISBN: 978-0-262-03384-8 and 978-0-262-53305-8.

Corwin, Edward and Antonette Logar (2004). "Sorting in Linear Time - Variations on the Bucket Sort". In: *Journal of Computing Sciences in Colleges* 20.1, pp. 197–202. ISSN: 1937-4771. URL: https://dl.acm.org/doi/abs/10.5555/1040231.1040257 (visited on 06/28/2023).

Culurciello, Eugenio and Andreas G Andreou (2003). "A Comparative Study of Access Topologies for Chip-Level Address-Event Communication Channels". In: *IEEE Transactions on Neural Networks* 14.5. DOI: 10.1109/TNN.2003.816385.

Datta, Gourav, Souvik Kundu, and Peter A. Beerel (2021). "Training Energy-Efficient Deep Spiking Neural Networks with Single-Spike Hybrid Input Encoding". In: *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. DOI: 10.1109/IJCNN52387.2021.9534306.

Davies, Donald W. (1999). "The Bombe a Remarkable Logic Machine". In: *Cryptologia* 23.2, pp. 108–138. DOI: 10.1080/0161-119991887793.

Davison, Andrew P., Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger (2009). "PyNN: a common interface for neuronal network simulators". In: *Front. Neuroinform.* 2.11. DOI: 10.3389/neuro.11.011.2008.

Dayan, Peter and L. F. Abbott (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. Cambride, Massachusetts: The MIT press. ISBN: 0-262-04199-5.

De Sensi, Daniele, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler (2020). "An In-Depth Analysis of the Slingshot Interconnect". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–14. DOI: 10.1109/SC41405.2020.00039.

Deng, Yangdong and W.P. Maly (2005). "2.5-dimensional VLSI system integration". In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 13.6, pp. 668–677. DOI: 10.1109/TVLSI.2005.848814.

Dennard, R.H., F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc (1974). "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5, pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

Diesmann, M., M.-O. Gewaltig, and A. Aertsen (1999). "Stable propagation of synchronous spiking in cortical neural networks". In: *Nature* 402, pp. 529–533.

Duato, Jose, Sudhakar Yalamanchili, and Lionel M Ni (2003). *Interconnection Networks. an Engineering Approach*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann. ISBN: 978-0-08-050899-3.

Eckert Jr, John Presper and John W Mauchly (1964). "Electronic Numerical Integrator And Computer". U.S. pat. 3120606. URL: https://worldwide.espacenet.com/patent/search?q=pn%3DUS3120606A.

Eggen, Roger and Maurice Eggen (2019). "Thread and process efficiency in python". In: *Proceedings of the international conference on parallel and distributed processing techniques and appli-*

cations (PDPTA). The Steering Committee of The World Congress in Computer Science, pp. 32–36. ISBN: 1-60132-508-8.

Electronic Visions(s), Heidelberg University (2022). *libnux*. URL: https://github.com/electronicvisions/libnux.

Electronic Visions(s), Heidelberg University (n.d.[a]). *calix*. Calibration routines for HICANN-X. URL: https://github.com/electronicvisions/calix.

Electronic Visions(s), Heidelberg University (n.d.[b]). *fisch*. FPGA Instruction Set Compiler for HICANN. URL: https://github.com/electronicvisions/fisch.

Electronic Visions(s), Heidelberg University (n.d.[c]). *flange*. Linking C++ software stacks with SystemVerilog using DPI. URL: https://github.com/electronicvisions/flange.

Electronic Visions(s), Heidelberg University (n.d.[d]). *grenade*. GRaph-based Experiment Notation And Data-flow Execution. URL: https://github.com/electronicvisions/grenade.

Electronic Visions(s), Heidelberg University (n.d.[e]). *halco*. URL: https://github.com/electronicvisions/halco.

Electronic Visions(s), Heidelberg University (n.d.[f]). *haldls*. URL: https://github.com/electronicvisions/haldls.

Electronic Visions(s), Heidelberg University (n.d.[g]). *hxcomm*. Low-level communication with HICANN-X. URL: https://github.com/electronicvisions/hxcomm.

Electronic Visions(s), Heidelberg University (n.d.[h]). *hxtorch*. PyTorch for BrainScaleS-2. URL: https://github.com/electronicvisions/hxtorch.

Electronic Visions(s), Heidelberg University (n.d.[i]). *librma*. Fork of EXTOLL's RMA Userspace Library. URL: https://github.com/electronicvisions/librma.

Electronic Visions(s), Heidelberg University (n.d.[j]). *pynn-brainscales*. PyNN for BrainScaleS-2. URL: https://github.com/electronicvisions/pynn-brainscales.

EXTOLL GmbH (2017a). *Technology Overview*. Company Website archived from the original. URL: https://web.archive.org/web/20170722133708/http://www.extoll.de/technology (visited on 08/23/2023).

EXTOLL GmbH (2017b). *Tourmalet (ASIC)*. Company Website archived from the original. URL: https://web.archive.org/web/20170717092946/http://www.extoll.de/products/tourmalet (visited on 08/23/2023).

Fernández, Eduardo et al. (2021). "Visual percepts evoked with an intracortical 96-channel microelectrode array inserted in human occipital cortex". In: *The Journal of Clinical Investigation* 131.23. DOI: 10.1172/JCI151331.

Fieres, J., A. Grübl, S. Philipp, K. Meier, J. Schemmel, and F. Schürmann (2004). "A Platform for Parallel Operation of VLSI Neural Networks". In: *Proc. of the 2004 Brain Inspired Cognitive Systems Conference (BICS2004)*. University of Stirling, Scotland, UK.

Friedmann, Simon (2013). "A New Approach to Learning in Neuromorphic Hardware". PhD thesis. Ruprecht-Karls-Universität Heidelberg. DOI: 10.11588/heidok.00015359. URL: http://archiv.ub.uni-heidelberg.de/volltextserver/15359/.

Friedmann, Simon (2015). *Omnibus On-Chip Bus*. forked from https://github.com/five-elephants/omnibus. URL: https://github.com/electronicvisions/omnibus.

Friedmann, Simon, Johannes Schemmel, Andreas Grübl, Andreas Hartel, Matthias Hock, and Karlheinz Meier (2017). "Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System". In: *IEEE Transactions on Biomedical Circuits and Systems* 11.1, pp. 128–142. ISSN: 1932-4545. DOI: 10.1109/TBCAS.2016.2579164.

Fröning, Holger (2015). "EXTOLL and Data Movements in Heterogeneous Computing Environments". In: *Sustained Simulation Performance 2014*. Springer International Publishing, pp. 127–139. ISBN: 978-3-319-10626-7. DOI: 10.1007/978-3-319-10626-7_11.

Fröning, Holger and Heiner Litz (2010). "Efficient hardware support for the Partitioned Global Address Space". In: *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–6. DOI: 10.1109/IPDPSW.2010.5470851.

Fröning, Holger, Mondrian Nüssle, Heiner Litz, Christian Leber, and Ulrich Brüning (2013). "On Achieving High Message Rates". In: *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pp. 498–505. DOI: 10.1109/CCGrid.2013.43.

Fukushima, K. (1988). "Neocognitron: A Hierarchical Neural Network Capable of Visual Pattern Recognition". In: *Neural Networks* 1, pp. 119–130.

Fukushima, K. and S. Miyake (1982). "Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position". In: *Pattern Recognition* 15(6), pp. 455–469.

Fukushima, K., S. Miyake, and T. Ito (1983). "Neocognitron: A neural network model for a mechanism of visual pattern recognition". In: *IEEE Transactions on Systems, Man and Cybernetics* SMC-13, pp. 826–834.

Fukushima, Kunihiko (1980). "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4, pp. 193–202.

Furber, Steve (2016). "Large-scale neuromorphic computing systems". In: *Journal of Neural Engineering* 13.5. DOI: 10.1088/1741-2560/13/5/051001.

Furber, Steve B., Francesco Galluppi, Steve Temple, and Luis A. Plana (2014). "The SpiNNaker Project". In: *Proceedings of the IEEE*. Vol. 102. 5, pp. 652–665. DOI: 10.1109/JPROC.2014.2304638.

Furber, Steve B., David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown (2013). "Overview of the SpiNNaker System Architecture". In: *IEEE Transactions on Computers* 62.12. ISSN: 0018-9340. DOI: 10.1109/TC.2012.142.

Geib, Benjamin Ulrich (2012). "Hardware support for efficient packet processing". PhD thesis. Mannheim University.

Gekle, Michael et al. (2015). *Taschenlehrbuch Physiologie*. 2., überarbeitete Auflage. Georg Thieme Verlag. ISBN: 978-3-13-144982-5.

Gerstner, Wulfram and Romain Brette (2009a). "Adaptive exponential integrate-and-fire model". In: *Scholarpedia* 4.6, p. 8427. DOI: 10.4249/scholarpedia.8427.

Gerstner, Wulfram and Romain Brette (2009b). "Adaptive exponential integrate-and-fire model". In: *Scholarpedia* 4.6, p. 8427. DOI: 10.4249/scholarpedia.8427. URL: http://

www.scholarpedia.org/article/Adaptive_exponential_integrate-and-fire_model.

Gerstner, Wulfram and Werner Kistler (2002). *Spiking Neuron Models. Single Neurons, Populations, Plasticity*. Cambridge University Press.

Gerstner, Wulfram, Andreas K Kreiter, Henry Markram, and Andreas VM Herz (1997). “Neural codes: firing rates and beyond”. In: *Proceedings of the National Academy of Sciences* 94.24, pp. 12740–12741.

Gewaltig, M. O., M. Diesmann, and A. Aertsen (2001). “Propagation of cortical synfire activity: survival probability in single trials and stability in the mean.” In: *Neural Netw* 14.6-7, pp. 657–673. ISSN: 0893-6080. URL: http://view.ncbi.nlm.nih.gov/pubmed/11665761.

Giese, Alexander, Benjamin Kalisch, and Mondrian Nüssle (2012). *RMA2 Specification*. Tech. rep. revision 2.0.4, CAG Confidential. Institue for Computer Engineering, Computer Architecture Group.

Grübl, Andreas (2007). “VLSI Implementation of a Spiking Neural Network”. Document No. HD-KIP 07-10. PhD thesis. Ruprecht-Karls-University, Heidelberg. URL: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=1788.

Grzybowski, Andrzej and Matthew H. Kaufman (2007). “Sir Charles Bell (1774-1842): contributions to neuro-ophthalmology”. In: *Acta Ophthalmologica Scandinavica* 85.8, pp. 897–901. DOI: 10.1111/j.1600-0420.2007.00972.x.

Gustafson, John L. (1988). “Reevaluating Amdahl’s Law”. In: *Commun. ACM* 31.5, pp. 532–533. ISSN: 0001-0782. DOI: 10.1145/42411.42415.

Güttler, Maurice Gilbert (2017). “Achieving a Higher Integration Level of Neuromorphic Hardware using Wafer Embedding”. PhD thesis. Ruprecht-Karls-Universität Heidelberg. DOI: 10.11588/heidok.00023723.

Hanser, Hartwig, Christine Scholtyssek, et al. (2000). *Ramón y Cajal*. Lexikon der Neurowissenschaft.

Hartel, Andreas (2016). “Implementation and Characterization of Mixed-Signal Neuromorphic ASICs”. PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Hebb, Donald O. (1949). *The Organization of Behaviour*. New York: Wiley.

Heeger, David (2000). *Poisson model of spike generation*. Handout. New York University, pp. 1–13. URL: https://www.cns.nyu.edu/~david/handouts/poisson.pdf.

Hessler, Neal A, Aneil M Shirke, and Roberto Malinow (1993). “The probability of transmitter release at a mammalian central synapse”. In: *Nature* 366.6455, pp. 569–572. DOI: 10.1038/366569a0.

Hewlett Packard Enterprise (2023). *HPE Slingshot Interconnect*. URL: https://www.hpe.com/de/de/compute/hpc/slingshot-interconnect.html (visited on 09/21/2023).

Hodgkin, Alan Lloyd and Andrew F. Huxley (1952). “A quantitative description of membrane current and its application to conduction and excitation in nerve.” In: *J Physiol* 117.4, pp. 500–544. ISSN: 0022-3751. URL: http://view.ncbi.nlm.nih.gov/pubmed/12991237.

Hornung, Moritz (2020). “Adapting the Cortical Microcircuit Model for the BrainScaleS-1 hardware”. Bachelor thesis. Universität Heidelberg.

Huang, Ya-Chi, Meng-Hsueh Chiang, Shui-Jinn Wang, and Jerry G. Fossum (2017). "GAAFET Versus Pragmatic FinFET at the 5nm Si-Based CMOS Technology Node". In: *IEEE Journal of the Electron Devices Society* 5.3, pp. 164–169. DOI: 10.1109/JEDS.2017.2689738.

IEEE (2013). "IEEE Standard for Test Access Port and Boundary-Scan Architecture". In: *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444. DOI: 10.1109/IEEESTD.2013.6515989.

IEEE (2018). *IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language*. Standard 1800-2017 (Revision of IEEE Std 1800-2012). Institute of Electrical and Electronics Engineers. DOI: 10.1109/IEEESTD.2018.8299595.

IEEE (2021). *International Roadmap For Devices And Systems™: More Moore*. URL: https://irds.ieee.org/editions/2021/more-moore (visited on 09/21/2023).

Irvine, M.M. (2001). "Early digital computers at Bell Telephone Laboratories". In: *IEEE Annals of the History of Computing* 23.3, pp. 22–42. DOI: 10.1109/85.948904.

ITU (1994). *ITU-T X.200 (07/1994)*. URL: https://handle.itu.int/11.1002/1000/2820.

Jones, Edward G. (1999). "Golgi, Cajal and the Neuron Doctrine". In: *Journal of the History of the Neurosciences* 8.2, pp. 170–178. DOI: 10.1076/jhin.8.2.170.1838.

Jurczak, M., N. Collaert, A. Veloso, T. Hoffmann, and S. Biesemans (2009). "Review of FINFET technology". In: *2009 IEEE International SOI Conference*, pp. 1–4. DOI: 10.1109/SOI.2009.5318794.

Kaiser, Jakob, Sebastian Billaudelle, Eric Müller, Christian Tetzlaff, Johannes Schemmel, and Sebastian Schmitt (2022). "Emulating dendritic computing paradigms on analog neuromorphic hardware". In: *Neuroscience* 489, pp. 290–300. ISSN: 0306-4522. DOI: 10.1016/j.neuroscience.2021.08.013. URL: https://www.sciencedirect.com/science/article/pii/S0306452221004218.

Kanzleiter, Lea (2018). "A Parametrizable Switch For Neuromorphic Hardware". Bachelor thesis. Ruprecht-Karls-Universität Heidelberg.

Karasenko, Vitali (2014). *A communication infrastructure for a neuromorphic system*. Master's thesis (English), University of Heidelberg.

Karasenko, Vitali (2020). "Von Neumann bottlenecks in non-von Neumann computing architectures". PhD thesis. Universität Heidelberg. URL: http://archiv.ub.uni-heidelberg.de/volltextserver/28691/1/KarasenkoPhD.pdf.

Kernighan, Brian W. and Dennis M. Ritchie (1977). *The M4 Macro Processor*. Tech. rep. Murray Hill, New Jersey 07974: Bell Laboratories.

Ketcham, Carl (2004). "Method and apparatus for packet aggregation in packet-based network". U.S. pat. 6721334b1. URL: https://patents.google.com/patent/US6721334B1/en.

Kiene, Gerd (2017). "Mixed-Signal Neuron and Readout Circuits for a Neuromorphic System". Master thesis. Ruprecht-Karls-Universität Heidelberg.

Kilby, Jack S. (1964). "Miniturized Electronic Circuits". U.S. pat. 3138743. URL: https://worldwide.espacenet.com/patent/search?q=pn%3DUS3138743A (visited on 06/12/2023).

Kim, Jinwoo et al. (2020). “Architecture, Chip, and Package Codesign Flow for Interposer-Based 2.5-D Chiplet Integration Enabling Heterogeneous IP Reuse”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 28.11, pp. 2424–2437. DOI: 10.1109/TVLSI.2020.3015494.

Kim, Kyungtae, S. Ganguly, R. Izmailov, and Sangjin Hong (2006). “On Packet Aggregation Mechanisms for Improving VoIP Quality in Mesh Networks”. In: *2006 IEEE 63rd Vehicular Technology Conference*. Vol. 2, pp. 891–895. DOI: 10.1109/VETECS.2006.1682953.

Kleider, Mitja (2017). “Neuron Circuit Characterization in a Neuromorphic System”. HD-KIP 17-135. PhD thesis. Universität Heidelberg. URL: http://www.kip.uni-heidelberg.de/Veroeffentlichungen/details.php?id=3657.

Kohútka, Lukáš (2022). “Efficiency of Priority Queue Architectures in FPGA”. In: *Journal of Low Power Electronics and Applications* 12.3. ISSN: 2079-9268. DOI: 10.3390/jlpea12030039.

Kohútka, Lukáš, Lukáš Nagy, and Viera Stopjaková (2018). “A Novel Hardware-Accelerated Priority Queue for Real-Time Systems”. In: *2018 21st Euromicro Conference on Digital System Design (DSD)*, pp. 46–53. DOI: 10.1109/DSD.2018.00023.

Kremkow, J., L.U. Perrinet, G.S. Masson, and A. Aertsen (2010). “Functional consequences of correlated excitatory and inhibitory conductances in cortical networks.” In: *J Comput Neurosci* 28, pp. 579–594.

Krol, Laurens R. (2021). *Action potential schematics.svg*. URL: https://commons.wikimedia.org/wiki/File:Action_potential_schematic.svg (visited on 09/19/2023).

Leibfried, Aron (2021). “On-chip calibration and closed-loop experiments on analog neuromorphic hardware”. Master thesis. Ruprecht-Karls-Universität Heidelberg.

Lenfant, Jaques (1978). “Parallel Permutations of Data: A Benes Network Control Algorithm for Frequently Used Permutations”. In: *IEEE Transactions on Computers* C-27.7, pp. 637–647. DOI: 10.1109/TC.1978.1675164.

Leonard, William R. and Marcia L. Robertson (1994). “Evolutionary perspectives on human nutrition: The influence of brain and body size on diet and metabolism”. In: *American Journal of Human Biology* 6.1, pp. 77–88. DOI: 10.1002/ajhb.1310060111.

Li, Hongmin, Hanchao Liu, Xiangyang Ji, Guoqi Li, and Luping Shi (2017). “CIFAR10-DVS: An Event-Stream Dataset for Object Classification”. In: *Frontiers in Neuroscience* 11. ISSN: 1662-453X. DOI: 10.3389/fnins.2017.00309.

Lichtsteiner, Patrick, Christoph Posch, and Tobi Delbruck (2008). “A 128$\times$128 120 dB 15 μs Latency Asynchronous Temporal Contrast Vision Sensor”. In: *IEEE Journal of Solid-State Circuits* 43, pp. 566–576. DOI: 10.1109/JSSC.2007.914337.

Litz, Heiner, Holger Fröning, Mondrian Nüssle, and Ulrich Brüning (2008). “VELO: A novel communication engine for ultra-low latency message transfers”. In: *2008 37th International Conference on Parallel Processing*, pp. 238–245. ISBN: 9780769533742. DOI: 10.1109/ICPP.2008.85.

Litz, Heiner Hannes (2011). “Improving the Scalability of High Performance Computer Systems”. PhD thesis. Mannheim University.

Liu, Min and Tobi Delbruck (2017). "Block-matching optical flow for dynamic vision sensors: Algorithm and FPGA implementation". In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–4. DOI: 10.1109/ISCAS.2017.8050295.

Mahowald, Misha (1992). "VLSI analogs of neuronal visual processing: a synthesis of form and function". PhD thesis. California Institute of Technology. DOI: 10.7907/Z9CZ35CD.

Markram, H. (2012). "The Human Brain Project". In: *Scientific American* 306.6, pp. 50–55.

Markram, H., J. Lübke, M. Frotscher, and B. Sakmann (1997). "Regulation of Synaptic Efficacy By Coincidence of Postsynaptic Aps." In: *Science* 275, pp. 213–215.

Mayr, Christian, Sebastian Hoeppner, and Steve Furber (2019). "Spinnaker 2: A 10 million core processor system for brain simulation and machine learning". In: *arXiv preprint arXiv:1911.02385*.

McKeown, N. (1999). "The iSLIP scheduling algorithm for input-queued switches". In: *IEEE/ACM Transactions on Networking* 7.2, pp. 188–201. DOI: 10.1109/90.769767.

Millner, Sebastian (2012). "Development of a Multi-Compartment Neuron Model Emulation". PhD thesis. Ruprecht-Karls-Universität Heidelberg. DOI: 10.11588/heidok.00013979.

Minsky, Marvin and Seymour Papert (1969). *Perceptrons. An Introduction to Computational Geometry*. Cambridge, MA: MIT Press.

Moore, Gordon E. (2006). "Lithography and the future of Moore's Law". In: *IEEE Solid-State Circuits Society Newsletter* 11.3, pp. 37–42. DOI: 10.1109/N-SSC.2006.4785861.

Müller, Eric, Elias Arnold, et al. (2022). "A Scalable Approach to Modeling on Accelerated Neuromorphic Hardware". In: *Front. Neurosci.* 16. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.884128.

Müller, Eric, Arne Emmel, et al. (2023). *The BrainScaleS-2 Neuromorphic Platform — A Report on the Integration and Operation of an Open Science Hardware Platform within EBRAINS*. DOI: 10.5281/zenodo.8375522.

Müller, Eric, Christian Mauch, Philipp Spilger, Oliver Julien Breitwieser, Johann Klähn, David Stöckel, Timo Wunderlich, and Johannes Schemmel (2020). *Extending BrainScaleS OS for BrainScaleS-2*. Tech. rep. Heidelberg, Germany: Electronic Vision(s), Kirchhoff Institute for Physics, Heidelberg University, Germany. DOI: 10.48550/arXiv.2003.13750. arXiv: 2003.13750 [cs.NE].

Müller, Eric, Moritz Schilling, and Christian Mauch (2018). *HostARQ Slow Control Transport Protocol*. URL: https://github.com/electronicvisions/sctrltp.

Müller, Eric, Sebastian Schmitt, et al. (2020). "The Operating System of the Neuromorphic BrainScaleS-1 System". In: *arXiv preprint*. submitted to Neurocomputing OSP. arXiv: 2003.13749 [cs.NE]. URL: http://arxiv.org/abs/2003.13749.

Najmaei, Sina, Andreu L. Glasmann, Marshall A. Schroeder, Wendy L. Sarney, Matthew L. Chin, and Daniel M. Potrepka (2022). "Advancements in materials, devices, and integration schemes for a new generation of neuromorphic computers". In: *Materials Today* 59, pp. 80–106. ISSN: 1369-7021. DOI: https://doi.org/10.1016/j.mattod.2022.08.017.

Nielsen, Bo Friis (2020). *Lecture notes on phase-type distributions for 02407 Stochastic Processes*. Technical University of Denmark, pp. 1–21. URL: http://www2.imm.dtu.dk/courses/02407/lectnotes/ftf.pdf.

Nobel Prize Outreach AB (2023). *All Nobel Prizes*. URL: https://www.nobelprize.org/prizes/lists/all-nobel-prizes/ (visited on 09/20/2023).

Noyce, Robert N. (1961). "Semiconductor Device-And-Lead Structure". U.S. pat. 2981877. URL: https://worldwide.espacenet.com/patent/search?q=pn%3DUS2981877A.

Nüßle, Mondrian Benediktus (2008). "Acceleration of the Hardware-Software Interface of a Communication Device For Parallel Systems". PhD thesis. Mannheim University.

Nüssle, Mondrian, Benjamin Geib, Holger Fröning, and Ulrich Brüning (2009). "An FPGA-based custom high performance interconnection network". In: *2009 International Conference on Reconfigurable Computing and FPGAs*. IEEE, pp. 113–118. DOI: 10.1109/ReConFig.2009.23.

Nüssle, Mondrian, Martin Scherer, and Ulrich Brüning (2009). "A Resource Optimized Remote-Memory-Access Architecture for Low-latency Communication". In: *International Conference on Parallel Processing*, pp. 220–227. DOI: 10.1109/ICPP.2009.62.

OCP (2009). *Open Core Protocol Specification 3.0*. URL: http://www.ocpip.org/home.

Ousterhout, John and Community (2023). *Tcl Developer Xchange*. Open Source Programming Language. URL: https://www.tcl.tk (visited on 06/01/2023).

Paszke, Adam et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

Pearson, Martin J, Shirin Dora, Oliver Struckmeier, Thomas C Knowles, Ben Mitchinson, Kshitij Tiwari, Ville Kyrki, Sander Bohte, and Cyriel Pennartz (2021). "Multimodal representation learning for place recognition using deep Hebbian predictive coding". In: *Frontiers in Robotics and AI* 8. DOI: 10.3389/frobt.2021.732023.

Pehle, Christian (2021). "Adjoint equations of spiking neural networks". PhD thesis. Ruprecht-Karls-Universität Heidelberg. DOI: 10.11588/heidok.00029866.

Pehle, Christian et al. (2022). "The BrainScaleS-2 Accelerated Neuromorphic System with Hybrid Plasticity". In: *Frontiers in Neuroscience* 16. ISSN: 1662-453X. DOI: 10.3389/fnins.2022.795876.

Peterson, Martin (2022). "The St. Petersburg Paradox". In: *The Standford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2022. Metaphysics Research Lab, Stanford University. URL: https://plato.stanford.edu/archives/sum2022/entries/paradox-stpetersburg/.

Petrovici, Mihai Alexandru (2016). *Form Versus Function. Theory and Models for Neuronal Substrates*. Springer Theses. Springer Cham, pp. XXVI, 374. ISBN: 978-3-319-39552-4. DOI: 10.1007/978-3-319-39552-4.

Philipp, Stefan (2008). "Design and Implementation of a Multi-Class Network Architecture for Hardware Neural Networks". PhD thesis. Ruprecht-Karls-Universität Heidelberg.

Piccolino, Marco (1998). "Animal electricity and the birth of electrophysiology: the legacy of Luigi Galvani". In: *Brain Research Bulletin* 46.5, pp. 381–407. ISSN: 0361-9230. DOI: 10.1016/S0361-9230(98)00026-4.

Plana, Luis A., Jim Garside, Jonathan Heathcote, Jeffrey Pepper, Steve Temple, Simon Davidson, Mikel Lujan, and Steve Furber (2020). “spiNNlink: FPGA-Based Interconnect for the Million-Core SpiNNaker System”. In: *IEEE Access* 8, pp. 84918–84928. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2020.2991038.

Potjans, Tobias C. and Markus Diesmann (2012). “The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Modela”. In: *Cereb. Cortex* 24 (3), pp. 785–806. DOI: 10.1093/cercor/bhs358.

Prades, Javier, Federico Silla, José Duato, Holger Fröning, and Mondrian Nüssle (2012). “A New End-to-End Flow-Control Mechanism for High Performance Computing Clusters”. In: *2012 IEEE International Conference on Cluster Computing*, pp. 320–328. DOI: 10.1109/CLUSTER.2012.15.

Privault, Nicolas (2018). *Understanding Markov Chains*. 2nd ed. 2018. Springer Undergraduate Mathematics Series. Singapore: Springer Singapore, XVII, 372 pages. ISBN: 978-981-13-0658-7. DOI: 10.1007/978-981-13-0659-4.

Rajkumar, Ajay and Michael D Turner (2008). “Packet aggregation for real time services on packet data networks”. U.S. pat. 7391769b2. URL: https://patents.google.com/patent/US7391769B2/en.

Reinagel, Pamela and R Clay Reid (2000). “Temporal coding of visual information in the thalamus”. In: *Journal of neuroscience* 20.14, pp. 5392–5400.

Rettig, Marco (2019a). “Characterizing the Event Interface of the HICANN-X”. Bachelor thesis. Ruprecht-Karls-Universität Heidelberg.

Rettig, Marco (2019b). *Verification of a Parameterizable JTAG Driver Module*. Internship report.

Rieke, F., D. Warland, R. de Ruyter van Steveninck, and W. Bialek (1997). *Spikes - Exploring the neural code.* MIT Press, Cambridge, MA.

Rodrigues, Joel J. P. C. and Paulo A. C. S. Neves (2010). “A survey on IP-based wireless sensor network solutions”. In: *International Journal of Communication Systems* 23.8, pp. 963–981. DOI: 10.1002/dac.1099.

Rosenblatt, F. (1958). “The Perceptron: a probabilistic model for information storage and organization in the brain”. In: *Psychological Review* 65, pp. 386–408.

Roser, Max and Hannah Ritchie (2020). *Moore’s Law Transistor Count 1970-2020.png*. URL: https://commons.wikimedia.org/wiki/File:Moore%27s_Law_Transistor_Count_1970-2020.png (visited on 09/21/2023).

Sale, Tony (2004). “Alan Turing at Bletchley Park in World War II”. In: *Alan Turing: Life and Legacy of a Great Thinker*. Ed. by Christof Teuscher. Springer Berlin Heidelberg, pp. 441–462. ISBN: 978-3-662-05642-4. DOI: 10.1007/978-3-662-05642-4_18.

Schemmel, J., J. Fieres, and K. Meier (2008). “Wafer-Scale Integration of Analog Neural Networks”. In: *Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN)*.

Schemmel, Johannes, Sebastian Billaudelle, Philipp Dauer, and Johannes Weis (2022). “Accelerated Analog Neuromorphic Computing”. DOI: 10.1007/978-3-030-91741-8_6.

Schemmel, Johannes, Daniel Brüderle, Andreas Grübl, Matthias Hock, Karlheinz Meier, and Sebastian Millner (2010). “A Wafer-Scale Neuromorphic Hardware System for Large-Scale Neural

Modeling”. In: *Proceedings of the 2010 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1947–1950. DOI: 10.1109/ISCAS.2010.5536970.

Schmidt, Albrecht (2002). “Ubiquitous Computing - Computing in Context”. PhD thesis. Lancaster University (United Kingdom).

Schmidt, Alexander (2017). “Design und Charakterisierung einer Routing-Schnittstelle für Neuromorphe Hardware”. written in German language. Bachelor thesis. Ruprecht-Karls-Universität Heidelberg.

Schmidt, Hartmut et al. (2023). “From Clean Room to Machine Room: Commissioning of the First-Generation BrainScaleS Wafer-Scale Neuromorphic System”. In: *arXiv preprint*. DOI: 10.1088/2634-4386/acf7e4. arXiv: 2303.12359 [cs.ET].

Schmitt, Juri (2017). “Accelerating Checkpoint/Restart Application Performance in Large-Scale Systems with Network Attached Memory”. PhD thesis. Ruprecht-Karls-Universität Heidelberg. DOI: 10.11588/heidok.00023800.

Scholze, S., H. Eisenreich, et al. (2011). “A 32 GBit/s Communication SoC for a Waferscale Neuromorphic System”. In: *Integration, the VLSI Journal*. in press. DOI: 10.1016/j.vlsi.2011.05.003.

Scholze, S., S. Henker, J. Partzsch, C. Mayr, and R. Schuffny (2010). “Optimized queue based communication in VLSI using a weakly ordered binary heap”. In: *Mixed Design of Integrated Circuits and Systems (MIXDES), 2010 Proceedings of the 17th International Conference*, pp. 316–320.

Schreiber, Korbinian (2021). “Accelerated neuromorphic cybernetics”. PhD thesis. Universität Heidelberg.

Seindal, René, François Pinard, Gary V. Vaughan, and Eric Blake (2021). *GNU M4, version 1.4.19. A powerful macro processor*. URL: https://www.gnu.org/software/m4/manual/m4.pdf (visited on 06/01/2023).

Shalf, John, Sudip Dosanjh, and John Morrison (2011). “Exascale Computing Technology Challenges”. In: *High Performance Computing for Computational Science – VECPAR 2010*. Springer Berlin Heidelberg, pp. 1–25. ISBN: 978-3-642-19328-6. DOI: 10.1007/978-3-642-19328-6_1.

Sigman, Karl (2016). *Expected Number of Visits of a Finite State Markov Chain to a Transient State*. Columbia University in the City of New York, pp. 1–3. URL: www.columbia.edu/~ks20/4106-18-Fall/Notes-Transient.pdf.

Silver, David, Aja Huang, et al. (2016). “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587, pp. 484–489.

Silver, David, Julian Schrittwieser, et al. (2017). “Mastering the game of go without human knowledge”. In: *Nature* 550.7676, pp. 354–359.

Silvestri, L et al. (2021). “Universal autofocus for quantitative volumetric microscopy of whole mouse brains”. In: *Nature Methods* 18.8, pp. 953–958. DOI: 10.1038/s41592-021-01208-1.

Sima, Dezsö (2000). “The design space of register renaming techniques”. In: *IEEE Micro* 20.5, pp. 70–83. DOI: 10.1109/40.877952.

Sivilotti, Massimo Antonio (1991). "Wiring Considerations in Analog VLSI Systems, with Application to Field-Programmable Networks". PhD thesis. California Institute of Technology. DOI: 10.7907/stj4-kh72.

Slogsnat, David, Alexander Giese, Mondrian Nüssle, and Ulrich Brüning (2008). "An Open-Source HyperTransport Core". In: *ACM Trans. Reconfigurable Technol. Syst.* 1.3. ISSN: 1936-7406. DOI: 10.1145/1391732.1391734.

Spilger, Philipp (2021). "From Neural Network Descriptions to Neuromorphic Hardware — A Signal-Flow Graph Compiler Approach". Master's thesis. Universität Heidelberg.

Spilger, Philipp, Elias Arnold, Luca Blessing, Christian Mauch, Christian Pehle, Eric Müller, and Johannes Schemmel (2023). "hxtorch.snn: Machine-learning-inspired Spiking Neural Network Modeling on BrainScaleS-2". In: *Neuro-inspired Computational Elements Workshop (NICE 2023)*. University of Texas, San Antonio, USA: Association for Computing Machinery, pp. 57–62. DOI: 10.1145/3584954.3584993. arXiv: 2212.12210 [cs.NE].

Stacho, Martin, Christina Herold, Noemi Rook, Hermann Wagner, Markus Axer, Katrin Amunts, and Onur Güntürkün (2020). "A cortex-like canonical circuit in the avian forebrain". In: *Science* 369.6511. DOI: 10.1126/science.abc5534.

Stevens, Charles and Anthony Zador (1995). "When is an integrate-and-fire neuron like a poisson neuron?" In: *Advances in neural information processing systems* 8.

Stradmann, Yannik and Johannes Schemmel (2023). "Biomorphic control for high-speed robotic applications". In: *7th HBP Student Conference on Interdisciplinary Brain Research*. Frontiers Media SA, pp. 277–281.

Straub, Jan Valentin (2023). "Multi-Single-Chip Training of Spiking Neural Networks with BrainScaleS-2". HD-KIP 23-53. Bachelor thesis. Ruprecht-Karls-Universität Heidelberg.

Swadlow, A. Harvey and Dr. Stephen G. Waxman (2012). "Axonal conduction delays". In: *Scholarpedia* 7 (6), p. 1451. ISSN: 1941-6016. DOI: 10.4249/scholarpedia.1451.

SystemVerilog (2004). *SystemVerilog 3.1a Language Reference Manual*. Accellera.

Takawadekar, Vaibhav and Yogita M. Vaidya (2021). "Neuromorphic computing: Modelling of 3D integrated circuit components using TSV". In: *2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, pp. 1–7. DOI: 10.1109/ICCCNT51525.2021.9579593.

Thanasoulis, Vasileios (2019). "Analysis and Development of a Communication Infrastructure for a Wafer-scale Neuromorphic System". PhD thesis. Technical University of Dresden.

Thommes, Tobias (2018). "Design and Implementation of an EXTOLL Network-Interface for the Communication FPGA in the BrainScaleS Neuromorphic Computing System". Master thesis. Ruprecht-Karls-Universität Heidelberg.

Thommes, Tobias, Sven Bordukat, Andreas Grübl, Vitali Karasenko, Eric Müller, and Johannes Schemmel (2022). "Demonstrating BrainScaleS-2 Inter-Chip Pulse Communication using EXTOLL". In: *Neuro-inspired Computational Elements Workshop (NICE '22), March 29 – April 1, 2022*. Virtual Event, USA: Association for Computing Machinery, pp. 98–100. ISBN: 9781450395595. DOI: 10.1145/3517343.3517376. arXiv: 2202.12122 [cs.AR].

Thommes, Tobias, Niels Buwen, Andreas Grübl, Eric Müller, Ulrich Brüning, and Johannes Schemmel (2021). "BrainScaleS Large Scale Spike Communication using Extoll". In: *2021 8th Neuro*

Inspired Computational Elements Workshop (NICE'2020). peer-reviewed extended abstract incl. paper presentation. arXiv: 2111.15296 [cs.AR].

Thommes, Tobias, Andreas Grübl, and Johannes Schemmel (2023). "Optimising Spike Throughput and Latency for Spike-Event Accumulation on Packet-Based Interconnection Networks". In: *7th HBP Student Conference on Interdisciplinary Brain Research*. Frontiers Media SA, pp. 258–264.

Top 500 List (2023). Top500. URL: https://www.top500.org/lists/top500/2023/06/ (visited on 09/21/2023).

Tsodyks, M. and H. Markram (1997). "The neural code between neocortical pyramidal neurons depends on neurotransmitter release probability". In: *Proceedings of the national academy of science USA* 94, pp. 719–723.

Wagner, Fabien B et al. (2018). "Targeted neurotechnology restores walking in humans with spinal cord injury". In: *Nature* 563.7729, pp. 65–71. DOI: 10.1038/s41586-018-0649-2.

Wang, Huifang E. et al. (2023). "Delineating epileptogenic networks using brain imaging data and personalized modeling in drug-resistant epilepsy". In: *Science Translational Medicine* 15.680. DOI: 10.1126/scitranslmed.abp8982.

Waxman, Stephen G and Harvey A Swadlow (1976). "Ultrastructure of visual callosal axons in the rabbit". In: *Experimental Neurology* 53.1, pp. 115–127. DOI: 10.1016/0014-4886(76)90287-9.

Wehr, Michael and Anthony M Zador (2003). "Balanced inhibition underlies tuning and sharpens spike timing in auditory cortex". In: *Nature* 426.6965, pp. 442–446.

Wenzel, Martin (2018). "An Extendable Environment for Control and Status Register File Generation". Master thesis. Ruprecht-Karls-Universität Heidelberg.

WikimediaUser (2006). *Synapse Illustration2 tweaked.svg*. URL: https://commons.wikimedia.org/wiki/File:Synapse_Illustration2_tweaked.svg (visited on 03/09/2023).

WikimediaUser (2015). *Metastability D-Flipflops.svg*. URL: https://commons.wikimedia.org/wiki/File:Metastability_D-Flipflops.svg (visited on 08/08/2023).

WikimediaUser (2019). *Neuron.svg*. URL: https://commons.wikimedia.org/wiki/File:Neuron.svg (visited on 03/08/2023).

Wikipedia (2023a). *10 µm process — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=10_%C2%B5m_process&oldid=1176108259 (visited on 09/21/2023).

Wikipedia (2023b). *3 nm process — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=3_nm_process&oldid=1175852042 (visited on 09/21/2023).

Wikipedia (2023c). *History of neuroscience — Wikipedia, The Free Encyclopedia*. URL: https://en.wikipedia.org/w/index.php?title=History_of_neuroscience&oldid=1169719273 (visited on 09/21/2023).

Wikipedia (2023d). *Z1 (Rechner) — Wikipedia, die freie Enzyklopädie*. URL: https://de.wikipedia.org/w/index.php?title=Z1_(Rechner)&oldid=237136306 (visited on 09/20/2023).

Wikipedia (2023e). *Zuse Z2 — Wikipedia, die freie Enzyklopädie*. URL: https://de.wikipedia.org/w/index.php?title=Zuse_Z2&oldid=235237282 (visited on 09/20/2023).

Wu, Jian, Steve Furber, and Jim Garside (2009). “A Programmable Adaptive Router for a GALS Parallel System”. In: *2009 15th IEEE Symposium on Asynchronous Circuits and Systems*, pp. 23–31. DOI: 10.1109/ASYNC.2009.17.

Wunderlich, Timo et al. (2019). “Demonstrating Advantages of Neuromorphic Computation: A Pilot Study”. In: *Frontiers in Neuroscience* 13, p. 260. ISSN: 1662-453X. DOI: 10.3389/fnins.2019.00260. URL: https://www.frontiersin.org/article/10.3389/fnins.2019.00260.

Xanthopoulos, Thucydides (2009). “Digital Delay Lock Techniques”. In: *Clocking in Modern VLSI Systems*. Springer US, pp. 183–244. ISBN: 978-1-4419-0261-0. DOI: 10.1007/978-1-4419-0261-0_6.

Xilinx (2011). *LogiCORE IP ChipScope Pro Integrated Logic Analyzer (ILA)*. Version v1.04a.

Xilinx Inc. (2018). *7 Series FPGAs GTX/GTH Transceivers. User Guide*. ug476 1.12.1.

Xilinx Inc. (2019). *7 Series FPGAs Memory Resources: User Guide*. ug473, pp. 1–88. URL: https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf (visited on 07/13/2023).

Yu, Qiang, Huajin Tang, Jun Hu, and Kay Chen Tan (2017). *Neuromorphic cognitive systems. a learning and memory centered approach*. Intelligent systems reference library. Cham: Springer International Publishing. ISBN: 978-3-319-55310-8.

Zimmermann, Hubert (1980). “OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection”. In: *IEEE Transactions on Communications* 28.4, pp. 425–432. DOI: 10.1109/TCOM.1980.1094702.

Zoschke, Kai, Maurice Guettler, Lars Boettcher, Andreas Gruebl, Dan Husmann, Johannes Schemmel, Karlheinz Meier, and Oswin Ehrmann (2017). “Full Wafer Redistribution and Wafer Embedding as Key Technologies for a Multi-Scale Neuromorphic Hardware Cluster”. In: *EPTC 2017*. DOI: 10.1109/EPTC.2017.8277579.

Acknowledgements - Danksagungen

Zum Schluss möchte ich allen Personen danken, die durch ihre freundliche Unterstützung zum Gelingen dieser Arbeit beigetragen haben. Ganz besonders danken möchte ich an dieser Stelle

- **Dr. habil. Johannes Schemmel** für die Übernahme der Betreuung und Begutachtung dieser Arbeit nach Karlheinz Meiers Tod und dafür, dass er mich immer in all meinen Ideen unterstützt hat.
- **dem verstorbenen Prof. Dr. Karlheinz Meier** dass er mir diese Doktorarbeit ermöglicht hat. Möge er bei Gott ewigen Frieden finden.
- **Prof. Dr. Peter Fischer** für die freundliche Übernahme des Zweitgutachtens.
- **Prof. Dr. em. Ulrich Brüning** für seine freundliche, stets kompetente und sehr hilfreiche Betreuung seit meiner Bachelorarbeit im Jahr 2015 über meine Masterarbeit im Jahr 2017-18 bis zu meiner Doktorarbeit. Ebenfalls vielen Dank für die vielen spannenden Vorlesungen, die immer gespickt waren mit erhellenden Anekdoten aus der Geschichte der Computerentwicklung. Möge die Macht stets mit Ihnen sein!
- **Prof. Dr. Wolfram Pernice** und **Prof. Dr. Matthias Bartelmann**, dass sie als Prüfer an der Verteidigung meiner Dissertation teilhaben werden.
- **Dr. Andreas Grübl** für viele fruchtbare Diskussionen zu den Themen dieser Arbeit und für seine stetige und teils aufopferungsvolle Mitbetreuung seit meiner Masterarbeit. Für seine Co-Autorenschaft bei meinen Veröffentlichungen und *last but not least* für die vielen hilfreichen Kommentare beim Korrekturlesen dieses Manuskripts.
- **Dr. Juri Schmidt** und **Dr. Vitali Karasenko** für die Co-Betreuung meiner Masterarbeit und ihre Unterstützung beim Zurechtfinden in den Details der BrainScaleS FPGA designs und der EXTOLL RMA Unit.
- **Niels Buwen, Leonard Henger** und **Sven Bordukat** für ihre Arbeit an der Software Integration des NHTL EXTOLL Transport-Layers.
- **Dr. Eric Müller, Dr. Christian Mauch und Phillip Spilger** für die Betreuung der Software Integration und ihre Unterstützung beim Zurechtfinden im BrainScaleS Softwarestack.
- **Joscha Ilmberger** für seine Expertise in vielen Aspekten der BrainScaleS-2 Hardware.

- **Dr. Niels Burkhardt, Dr. Dirk Frey, Tobias Groschup, Dr. Benjamin Kalisch, Dr. Sarah Neurwirth und Dr. Mondrian Nüssle** für ihre Unterstützung seitens der EXTOLL GmbH.
- **Dr. Andreas Baumbach, Dr. Sebastian Billaudelle, Julian Gölz, Dr. Andreas Grübl, Jakob Kaiser, Phillip Spilger und Yannik Stradmann** für das Korrekturlesen dieses Manuskripts.
- meinen ehemaligen und aktuellen Büro-Kollegen in chronologischer Reihenfolge **Dr. Sebastian Billaudelle, Dr. Christian Pehle, Dr. Korbinian Schreiber, Dr. Benjamin Kramer, Simeon Kanya, Yannik Stradmann, Phillip Spilger, Phillip Dauer, Johannes Weis, Robin Heinemann** und **Kaspar Haas** für viele entspannte und produktive Stunden am Arbeitsplatz, die durch ihre freundliche, professionelle und oft erheiternde Gesellschaft, die Arbeit nie haben langweilig werden lassen.
- der **Kicker-Crew**, hier besonders **Sebastian, Joscha, Yannik, Phillip S. und Phillip D. und Eric** für viele lustige Momente und legendäre (Eigen-) Tore in alle Richtungen.
- **Dr. Christian Mauch** für seine Grillkünste im Dienste aller Visions.
- **allen Visions** und den **Mitgliedern der ehemaligen Rechnerarchitektur Gruppe** für die tolle Zusammenarbeit und die schöne gemeinsame Zeit.
- meinen Feuerwehr Kameradinnen und Kameraden in Langenthal und Neuenheim für viele gemeinsame Festbesuche, Übungs- und Freizeitstunden, in denen ich meine Arbeit beiseitelegen und geistige Kraft tanken konnte.
- meiner Tanzpartnerin **Lea Gammelin** für die vielen tänzerisch anspruchsvollen und schönen, teils lustigen Stunden.
- meinen Eltern **Dr. Eduard Thommes** und **Maria Thommes**, sowie meinen Großeltern **Ewald und Franziska Thommes** und **Alfred und Elisabeth Tures** für ihre Liebe und Unterstützung in allen Lebenslagen.
- meinem Bruder **Johannes** und meiner Schwester **Anna** für viele tolle gemeinsame Kindheitserinnerungen und Erlebnisse. Ein besonderer Dank geht an **Anna**, die mir in den letzten Wochen meiner Arbeit einen Grund zum Frühaufstehen gegeben hat ;-).

Statement of originality - Erklärung:

I certify that this thesis, and the research to which it refers, are the product of my own work. Any ideas or quotations from the work of other people, published or otherwise, are fully acknowledged in accordance with the standard referencing practices of the discipline.

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 4. Oktober 2023 . . . .