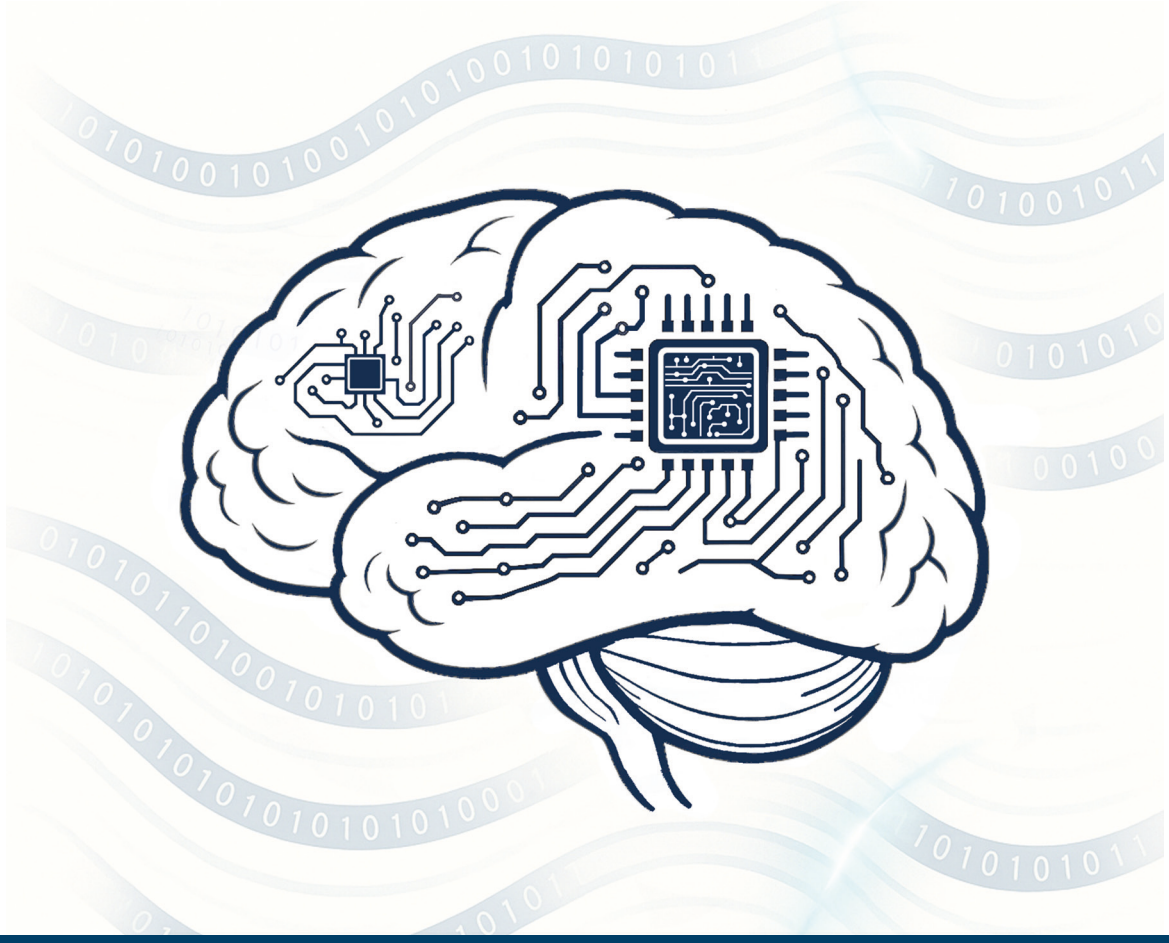


Development and Evaluation of Architecture Concepts for a System-on-Chip Based Neuromorphic Compute Node for Accelerated and Reproducible Simulations of Spiking Neural Networks in Neuroscience

Guido Trensch

IAS Series
Band / Volume 71
ISBN 978-3-95806-832-2

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
Forschungszentrum

Schriften des Forschungszentrums Jülich
IAS Series Band / Volume 71

Forschungszentrum Jülich GmbH
Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)

Development and Evaluation of Architecture Concepts for a System-on-Chip Based Neuromorphic Compute Node for Accelerated and Reproducible Simulations of Spiking Neural Networks in Neuroscience

Guido Trensch

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 71

ISSN 1868-8489

ISBN 978-3-95806-832-2

Bibliografische Information der Deutschen Nationalbibliothek.
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der
Deutschen Nationalbibliografie; detaillierte Bibliografische Daten
sind im Internet über http://dnb.d-nb.de abrufbar.

Herausgeber und Vertrieb:	Forschungszentrum Jülich GmbH Zentralbibliothek, Verlag 52425 Jülich Tel.: +49 2461 61-5368 Fax: +49 2461 61-6103 zb-publikation@fz-juelich.de www.fz-juelich.de/zb
Umschlaggestaltung:	Grafische Medien, Forschungszentrum Jülich GmbH
Druck:	Grafische Medien, Forschungszentrum Jülich GmbH
Copyright:	Forschungszentrum Jülich 2025

Schriften des Forschungszentrums Jülich
IAS Series, Band / Volume 71

D 82 (Diss. RWTH Aachen University, 2024)

ISSN 1868-8489
ISBN 978-3-95806-832-2

Vollständig frei verfügbar über das Publikationsportal des Forschungszentrums Jülich (JuSER)
unter www.fz-juelich.de/zb/openaccess.

Abstract

Despite the great strides neuroscience has made in recent decades, the underlying principles of brain function remain largely unknown. Advancing the field strongly depends on the ability to study large-scale neural networks and perform complex simulations. Simulations in hyper-real time are of high interest here, as they would enable both comprehensive parameter scans and the study of slow processes such as learning and long-term memory. Not even the fastest supercomputer available today is capable of meeting the challenge of accurate and reproducible simulation with hyper-real acceleration. The development of novel neuromorphic computing architectures holds out promise, but the high costs and long development cycles for application-specific hardware solutions makes it difficult to keep pace with the rapid developments in neuroscience. Commercial off-the-shelf System-on-Chip (SoC) devices, integrating programmable logic, general-purpose processors, and memory in a single chip, offer an alternative. This technology is providing interesting new design possibilities for application-specific implementations while avoiding costly chip development.

The primary aim of this thesis is to develop and evaluate a novel SoC-based architecture for a neuromorphic compute node intended to operate in a multi-node cluster configuration and capable of performing hyper-real-time simulations. As a complementary, yet distinct approach to the neuromorphic developments aiming at brain-inspired and highly efficient novel computing architectures for solving real-world tasks, the design of the compute node is strictly driven by neuroscience requirements. These requirements are demanding, as is the process of deriving appropriate design decisions from them.

Even for domain experts, it is often difficult to judge the correctness of a simulation result. This leaves some uncertainty when making design decisions and proving the correctness of an architectural design and its physical implementation. Methods for building credibility, such as verification and validation, have been developed but are not yet well established in the field of neural network modeling and simulation. This thesis therefore also outlines a rigorous model substantiation methodology for increasing the correctness of neural network simulation results in the absence of experimental validation data. The method was applied during the development and

evaluation of the neuromorphic compute node to build credibility on implementation correctness.

Finally, with the goal of large-scale neuromorphic computing, related technological aspects are discussed and architectural enhancements for the neuromorphic compute node are presented. This is accompanied by a workload analysis of two large-scale neural network models used in neuroscience. Also, a concept for system integration is proposed that incorporates the high-performance computing (HPC) landscape and takes into account existing tools and workflows for modeling and simulation in computational neuroscience.

The results presented in this thesis reveal the potential of commercial off-the-shelf SoC technology and demonstrate its suitability as a substrate for neuromorphic computing for application in computational neuroscience. Recent developments in this technology, particularly the integration of high-bandwidth memory (HBM), promise significant performance improvements. Acceleration factors on the order of 100 become within reach, even for the simulation of large-scale spiking neural networks.

Zusammenfassung

Trotz der enormen Fortschritte, die die Neurowissenschaften in den letzten Jahrzehnten erzielt haben, sind die grundlegenden Prinzipien der Funktionsweise des Gehirns noch weitgehend unverstanden. Der Fortschritt auf diesem Gebiet hängt stark von der Fähigkeit ab, großskalige neuronale Netzwerke untersuchen zu können und komplexe Simulationen durchzuführen. In diesem Zusammenhang sind Simulationen in Hyper-Echtzeit von großem Interesse, da dies sowohl umfassende Parameterscans als auch das Studium langsamer Prozesse, wie Lernen und Langzeitgedächtnis, ermöglichen würde. Doch selbst der leistungsfähigste heute verfügbare Supercomputer ist nicht in der Lage, die Herausforderung einer genauen, reproduzierbaren und zugleich signifikant beschleunigten Simulation zu bewältigen. Die Entwicklung neuartiger neuromorpher Computerarchitekturen ist hier vielversprechend. Dem gegenüber stehen jedoch hohe Kosten und lange Entwicklungszyklen für anwendungsspezifische Hardwarelösungen, die es erschweren, mit dem rasanten Tempo der Entwicklungen in den Neurowissenschaften Schritt zu halten. Eine Alternative bietet hier kommerziell verfügbare System-on-Chip (SoC) Technologie, die programmierbare Logik, Allzweckprozessoren und Speicher in einem einzigen Chip integriert. Diese Technologie eröffnet interessante neue Designmöglichkeiten für anwendungsspezifische Implementierungen, wobei eine kostspielige Chipentwicklung vermieden wird.

Das Hauptziel dieser Arbeit ist die Entwicklung und Evaluierung einer neuartigen SoC-basierten Architektur eines neuromorphen Rechenknotens, der in einer Clusterkonfiguration betrieben werden soll und in der Lage ist, Hyper-Echtzeit-Simulationen durchzuführen. Als komplementärer, aber dennoch eigenständiger Ansatz zu den neuromorphen Entwicklungen, die auf vom Gehirn inspirierte und hocheffiziente neuartige Computerarchitekturen zur Lösung realer Aufgaben abzielen, orientiert sich das Design des Rechenknotens streng an den Anforderungen der Neurowissenschaften. Diese Anforderungen sind anspruchsvoll, ebenso wie der Prozess der Ableitung angemessener Designentscheidungen daraus.

Selbst für Fachexperten ist es oft schwierig, die Korrektheit eines Simulationsergebnisses zu beurteilen. Dies führt zu einer Unsicherheit bei Designentscheidungen und beim Nachweis der Korrektheit eines Architekturentwurfes und dessen technischer Implementierung. Methoden, die

Sicherheit schaffen, wie z.B. Verifizierungs- und Validierungsverfahren, wurden zwar entwickelt, sind aber in den Neurowissenschaften im Bereich der Modellierung und Simulation neuronaler Netzwerke noch nicht gut etabliert.

Ergänzend stellt diese Arbeit daher eine strenge Methodologie zur Modellabsicherung vor, mit der die Korrektheit von Ergebnissen neuronaler Netzwerksimulationen erhöht werden kann, wenn keine experimentellen Validierungsdaten zur Verfügung stehen. Bei der Entwicklung und Evaluierung des neuromorphen Rechenknotens wurde diese Methodik eingesetzt, um Sicherheit hinsichtlich der Implementierungskorrektheit zu schaffen.

Schliesslich, mit dem Ziel großskaligen neuromorphen Computings, werden hierbei relevante technologische Aspekte diskutiert und Architekturverbesserungen für den neuromorphen Rechenknoten aufgezeigt. Begleitet wird dies von einer Analyse der bei der Simulation großskaliger Netwerke zu erwartenden Arbeitslast. Hierfür werden zwei in den Neurowissenschaften verwendete großskalige neuronale Netzwerkmodelle herangezogen. Ebenso wird ein Konzept für eine Systemintegration vorgestellt, welches die High-Performance Computing (HPC) Landschaft einbezieht und bestehende Werkzeuge und Arbeitsabläufe für die Modellierung und Simulation berücksichtigt.

Die Ergebnisse dieser Arbeit zeigen das Potential kommerziell verfügbarer SoC-Technologie und demonstrieren dessen Eignung als Platform für neuromorphes Computing für die Anwendung in den computergestützten Neurowissenschaften. Aktuelle Entwicklungen dieser Technologie, insbesondere die Integration von High-Bandwidth-Memory (HBM), versprechen darüber hinaus deutliche Leistungssteigerungen. Selbst für die Simulation großskaliger spikender neuronaler Netzwerke rücken damit Beschleunigungsfaktoren in der Größenordnung von 100 in den Bereich des technologisch Erreichbaren.

Acknowledgments

This thesis summarizes my work from 2017 to 2023 at the Research Center Jülich. Completing this thesis would not have been possible without the support of friends and colleagues, and I am very grateful to everyone I had the pleasure to work with during this project.

First and foremost, I would like to express my special thanks to my supervisor, Prof. Abigail Morrison, for always providing invaluable guidance and feedback. I am also indebted to Prof. Tobias Gemmeke for supporting my work and for taking on the role of the second reviewer. I furthermore would like to sincerely thank Prof. Matthias Müller and Prof. Torsten Kuhlen for taking on the roles of the third examiner and the chair of the examination committee, respectively.

I gratefully acknowledge all the colleagues I had the privilege to work with in the Advanced Computing Architectures (ACA) project. I cannot emphasize enough the value of this collaboration, the fruitful discussions we have had, and the feedback I have received. In particular, I would like to extend my special thanks to Prof. Tobias Noll, Dr. Georgia Psycho, Eqbal Maraqa, Dr. Michael Schiek, Dr. Markus Robens, Darshana Amarasingha, and especially Dr. Arne Heittmann for his comments on an earlier version of the manuscript on the neuromorphic compute node concept. I would also like to thank Prof. Markus Diesmann, one of the main initiators and driving forces of the ACA project, as well as Dr. Tom Tetzlaff for the inspiring discussions we had during our *neuromorphic tea* sessions and in his garden. I particularly remember our conversation about the computational cost of time- and event-driven simulations, which inspired the development of the workload and performance model.

I would also like to express my gratitude to Robin Gutzen, Dr. Michael Denker, Dr. Michael von Papen, Pietro Quaglio, and Prof. Sonja Grün for the fruitful collaboration throughout the model validation project. I would especially like to thank Fahad Kahlid for his comments on an earlier manuscript about model validation. I thank my colleagues at the Simulation and Data Laboratory Neuroscience and the Jülich Supercomputing Centre for their support and continuous collaboration, and for being a great bunch of people in and out of the lab. I also thank the NEST and NESTML developer community. Working in this community has given me valuable knowledge about modeling and simulation technologies. On a personal level, here I would like to

thank Dennis Terhorst, Dr. Jochen Martin Eppler and Dr. Dimitri Plotnikov.

I am also grateful to the German and European institutions that have supported my work. This project has received funding from the Helmholtz Association Initiative and Networking Fund under project No. SO-092 (Advanced Computing Architectures, ACA) and through the Helmholtz Portfolio Theme Supercomputing and Modeling for the Human Brain (SMHB). In addition, this project has also received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under Specific Grant Agreements No. 720270 (Human Brain Project SGA1) and No. 785907 (Human Brain Project SGA2). Open Access publications have been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 491111487.

Finally, I dedicate this work to my dear wife, Dr. phil. Nosrat Kazemi-Trensch, for her constant support, love, understanding, and encouragement through life's challenges.

Jülich, December 2024

Eidesstattliche Erklärung

Ich, Guido Trensch, erkläre hiermit, dass diese Dissertation und die darin dargelegten Inhalte die eigenen sind und selbstständig, als Ergebnis der eigenen originären Forschung, generiert wurden. Hiermit erkläre ich an Eides statt:

1. Diese Arbeit wurde vollständig oder größtenteils in der Phase als Doktorand und wissentschaflicher Mitarbeiter des Simulation and Data Laboratory Neuroscience des Forschungszentrum Jülich angefertigt.

2. Sofern irgendein Bestandteil diseser Dissertation zuvor für einen akademischen Abschluss oder eine andere Qualifikation an dieser oder einer anderen Institution verwendet wurde, wurde dies klar angezeigt.

3. Wenn immer andere, eigene oder Veröffentlichungen Dritter herangezogen wurden, wurden diese klar benannt.

4. Wenn aus anderen, eigenen oder Veröffentlichungen Dritter zitiert wurde, wurde stets die Quelle hierfür angegeben.

5. Diese Dissertation is vollständig meine eigene Arbeit, mit der Ausnahme solcher Zitate.

6. Alle wesentlichen Quellen von Unterstützung wurden benannt.

7. Wenn immer ein Teil dieser Dissertation auf der Zusammenarbeit mit anderen basiert, wurde von mir klar gekennzeichnet, was von anderen und was von mir selbst erarbeitet wurde.

8. Teile dieser Arbeit wurden zuvor veröffentlicht, ersichtlich im Abschnitt *Declaration of Publications and Contributions*.

Declaration of Publications and Contributions

Parts of this thesis have already been published in peer-reviewed scientific journals.

Guido Trensch, Robin Gutzen, Inga Blundell, Michael Denker, Abigail Morrison (2018). Rigorous Neural Network Simulations: A Model Substantiation Methodology for Increasing the Correctness of Simulation Results in the Absence of Experimental Validation Data. Frontiers in Neuroinfor- matics 12, 81. doi:10.3389/fninf.2018.00081

Author contributions:
The author devised the project, developed the main conceptual ideas and workflows, performed the verification tasks, the implementation of the models and their refinement, and carried out the numerical simulations under the supervision of Abigail Morrison. Inga Blundell contributed expertise on numeric integration. Robin Gutzen, Michael Denker, Abigail Morrison, and the author established the terminology. Robin Gutzen and Michael Denker designed the statistical analysis, which was then performed by Robin Gutzen. Robin Gutzen and Michael Denker have also written the passage on analysis of network spiking activity. The author has revised this section for this thesis (Section 2.3.2.2) and added descriptions of the statistical measures used throughout this work. The figures showing the results of the model substantiation assessments have been created by Robin Gutzen and are used in this thesis (Figures 2.11, B1 – B5). All other figures are the author's own work. Chapter 2 of this thesis is based on this publication. Here, some passages are quoted verbatim, particularly passages defining and describing terminology.

Robin Gutzen, Michael von Papen, Guido Trensch, Pietro Quaglio, Sonja Grün, Michael Denker (2018). Reproducible Neural Network Simulations: Statistical Methods for Model Validation on the Level of Network Activity Data. Frontiers in Neuroinformatics 12, 90. doi:10.3389/fninf.2018.00090

Author contributions:
Robin Gutzen, Michael von Papen, Sonja Grün, Michael Denker and the author designed the

study. Robin Gutzen and Pietro Quaglio performed the analysis. The author performed the simulations and implemented the model. Robin Gutzen, Michael von Papen, and Pietro Quaglio wrote the software for performing the validations. All authors contributed to the writing of the manuscript. The project was supervised by Sonja Grün and Michael Denker. This study accompanied and complemented the above publication.

Guido Trensch, Abigail Morrison (2022). A System-on-Chip Based Hybrid Neuromorphic Compute Node Architecture for Reproducible Hyper-Real-Time Simulations of Spiking Neural Networks. Frontiers in Neuroinformatics 16. doi:10.3389/fninf.2022.884033

Author contributions:

The author developed the System-on-Chip based hybrid neuromorphic compute node architecture, implemented the prototype, performed the hardware and software development, developed the workload and performance model, and carried out the experiments. The project was supervised by Abigail Morrison. Chapter 3 of this thesis is based on the architectural description presented in this publication. For this thesis, the author reworked and substantially expanded the content. The workload and performance model, and the results presented in this publication also formed the basis of Chapter 5.

Arne Heittmann, Georgia Psychou, Guido Trensch, Charls E. Cox, Winfried W. Wilcke, Markus Diesmann, Tobias G. Noll (2022). Simulating the Cortical Microcircuit Significantly Faster Than Real Time on the IBM INC-3000 Neural Supercomputer. Frontiers in Neuroscience doi:10.3389/fnins.2021.728460

Author contributions:

Arne Heittmann elaborated the concept of configurable spiking neural network (CsNN) simulations in cooperation with Tobias G. Noll, implemented and verified CsNN on the IBM Neural Computer INC-3000. Winfried W. Wilcke founded and leads the *Machine Intelligence* project at IBM Research, where the INC-3000 was designed and built. Charls E. Cox was the main hardware designer of the INC-3000 system. Georgia Psychou contributed by providing critical review, feedback, and proofreading of the manuscript. Markus Diesmann contributed to the initial draft and edited the manuscript. The author of this thesis has contributed to the validation and to the writing of the manuscript with the section on correctness. This section's description of statistical measures was revised and expanded for this thesis and is used in Section 2.3.2.2.

During the course of this dissertation, the author co-authored a publication that is related to this work but was not used in this thesis.

Inga Blundell, Romain Brette, Thomas A. Cleland, Thomas G. Close, Daniel Coca, Andrew P. Davison, Sandra Diaz-Pier, Musoles Carlos Fernandez, Padraig Gleeson, Dan F. M. Goodman, Hines Michael Michael, Michael W. Hopkins, Pramod Kumbhar, David R. Lester, Bóris Marin, Abigail Morrison, Eric Müller, Thomas Nowotny, Alexander Peyser, Dimitri Plotnikov, Paul Richmond, Andrew Rowley, Bernhard Rumpe, Marcel Stimberg, Alan B. Stokes, Adam Tomkins, Guido Trensch, Marmaduke Woodman, Jochen Martin Eppler (2018). Code Generation in Computational Neuroscience: A Review of Tools and Techniques. Frontiers in Neuroinformatics 12, 68. doi:10.3389/fninf.2018.00068

Author contributions:
Inga Blundell, Abigail Morrison, Jochen Martin Eppler, and Dimitri Plotnikov have coordinated the work on this review paper. All authors contributed to the publication with sections from their respective areas of expertise. The author of this thesis has written passages on classical processors and accelerators.

Contents

Chapter 1

Introduction

1.1 Motivation

Over the past century, an enormous body of knowledge has been accumulated about the structure and function of the nervous system and the brain. Our understanding of neurobiology has become increasingly precise and has been incorporated into neuroscience, which evolved into an academic discipline in its own right. In 1952, Alan Lloyd Hodgkin and Andrew Huxley presented a mathematical model that describes how action potentials in neurons of the giant axon of a squid are initiated and propagated (Hodgkin and Huxley, 1952). The *Hodgkin-Huxley* model is central to neuroscience research, and the authors received the 1963 Nobel Prize in Physiology or Medicine for their work. In 1961, Richard FitzHugh suggested a simplified version of this model, replacing the four equations of Hodgkin and Huxley by two (FitzHugh, 1961). An equivalent electrical circuit was presented in 1962 by Jinichi Nagumo (Nagumo et al., 1962). The model is therefore known as the *FitzHugh-Nagumo* model. In the same year, Bernard Katz modeled neurotransmission across nerve cells (neurons) and uncovered fundamental properties of synapses, the junctions between nerve cells that allow a signal to pass from one neuron to another. In the 1960s and 1970s, David Hubel and Torsten Wiesel greatly expanded our knowledge of sensory processing and the development of the visual system in a number of fundamental studies (Hubel and Wiesel, 2005). Beginning in 1966, Eric Kandel and collaborators studied biochemical changes in neurons associated with learning and memory in Aplysia and identified the physiological changes that occur in the brain during the formation and storage of memories (Kandel, 2007). How learning and information storage is achieved in the brain is still one of the central questions to neuroscience. Research in this area is heavily influenced by Hebb's 1949 postulate (Hebb, 1949), which states that a correlated activity of two neurons strengthens their synaptic connection.

During the second half of the twentieth century, advancements in molecular biology and technical innovations such as patch-clamp electrophysiology significantly increased scientific study (Altimus et al., 2020). In particular, the incredible technological progress in computer technology we have seen over the past five decades has accelerated neuroscience research. Computerized models of neurons and neural networks have made significant contributions here – and continue to do so. Computational neuroscience is still a young but rapidly growing area within the field of neuroscience, involving a variety of disciplines including mathematics, physics, computer science, and engineering. Supercomputers, with their unprecedented computational power, have become indispensable tools in today's neuroscience research with a broad range of applications. For example, in neuroimaging, which aims to understand the microscopic organization and detailed anatomy of the brain, supercomputers are used to analyze big data

sets (e.g., Axer and Amunts, 2022). Research on the principles of brain structure, dynamics, and function uses supercomputers to simulate large neural networks – networks with millions of neurons and billions of synapses.

Despite the great strides neuroscience has made in recent decades, the underlying principles of how the brain works are still largely unknown. Chris Eliasmith notes in his book *How to Build a Brain*: *"An answer that picks out the parts of the brain that have increased activity while reading words, or an answer that describes what chemicals are in less abundance when someone is depressed, is not what we have in mind. Certainly these observations are all part of an answer, but none of them traces the path from perception to action ..."* (Eliasmith, 2013). The ultimate goal remains to understand consciousness and the underlying mechanisms by which we perceive, learn, act, and remember.

The human brain contains nearly 86 billion neurons, with each neuron in the neocortex forming on the order of 10,000 synapses with other neurons. While the human brain consumes about the same amount of power as a light bulb, the power consumption of a supercomputer is five orders of magnitude larger, in the megawatt range. The von Neumann architecture of today's computers physically separates computation from storage. In brains, no conceptual or physical distinction is made here. Data processing and storage are performed in a distributed manner by complex networks of nerve cells. The brain's efficiency, computational capabilities, processing speed, and robustness to noise and malfunction is unique. From the earliest days of computing, this encouraged engineers to profitably apply brain principles in the design of machines. An early example is the *Mark I Perceptron* machine, built in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt (Cornell Aeronautical Laboratory Inc., 1960). Because of his pioneering work on artificial neural networks, Frank Rosenblatt is today recognized as one of the fathers of deep learning (Tappert, 2019).

Since then, a plethora of brain-inspired computers and devices have been built. They are used to model neuroscience theories and to solve machine learning tasks. The field of neuromorphic computing, as we are calling it today, is evolving at an incredible pace, where the research on neuromorphic algorithms and applications is becoming more and more of an important area. The term *neuromorphic* emerged in the late 1980s and it is closely associated with Carver Mead, a pioneer of modern microelectronics, who popularized the term with his 1990 publication entitled *Neuromorphic Electronic Systems* (Mead, 1990). Carver Mead also published the first book on neuromorphic computing, in which he describes electronic analog circuits to mimic neurobiological architectures found in the nervous system (Mead, 1989). Today, the term *neuromorphic computing* or *neuromorphic engineering* is much more broadly defined. We use it to refer to brain-inspired analog circuits and devices in the tradition of Carver Mead, such

as the neuron circuits developed by Giacomo Indiveri and his group (Indiveri et al., 2011), as well as advanced computer architectures that are inspired by brain principles or are dedicated to neuroscience simulation, such as the SpiNNaker neuromorphic system (Furber and Bogdan, 2020).

The main driver for most neuromorphic developments is undoubtedly brain-inspired computing aiming at solving real-world tasks, where it draws inspiration from insights from neuroscience. The application of neuromorphic computing in neuroscience research, however, is a niche area, although there is great interest in its application in modeling and simulation. Progress in neuroscience research depends to a large extent on the ability to study large neural networks and perform complex simulations. Performing simulations in hyper-real time is of great interest here, as it would allow comprehensive parameter scans and the study of slow processes such as learning and long-term memory. However, even the fastest and most advanced supercomputers available today cannot meet the challenge of significantly accelerating the simulation of a large-scale network. Neuromorphic computing, leveraging novel technologies and application-specific hardware architectures, is therefore an attractive option that promises to provide the necessary tools for this task.

Despite all technological innovations, making neuromorphic computing a useful tool for neuroscientists is a demanding technical challenge. Neuroscience research employs mathematical models to gain understanding of the complex dynamics of neural networks. Their simulation requires numerical accuracy. Judging the correctness of a simulation outcome is often difficult, even for domain experts. Methods for building credibility, such as verification and validation, have been developed and are common practice in engineering disciplines, but they are not yet well established in the field of neural network modeling and simulation. Deriving appropriate design decisions for the implementation of numerical operations and algorithms can be hampered by this. Neuroscience simulation also uses a wide variety of neuron and synapse models. Besides the requirement for accuracy, there is therefore also a need for flexibility. At a technical level, flexibility conflicts with the objective of achieving efficiency, which is tied to the goal of accelerated simulation. In addition, a neuromorphic system dedicated to neuroscience must also integrate with the existing landscape of tools and workflows for modeling and simulation in order to attain user acceptance.

To date, there is no neuromorphic system available to neuroscientists that can meet all criteria equally while achieving significant acceleration. A system capable of speeding up the simulation of a large-scale neural network by a factor of 100 with respect to the biological time domain would be a major breakthrough.

Developing such a system based on application-specific integrated circuits (ASICs) is a

time-consuming and costly process. A promising alternative is commercial off-the-shelf chip technology, integrating programmable logic, such as field-programmable gate arrays (FPGAs), along with general-purpose processors and memory in a single device called a System-on-Chip (SoC). This technology has the potential to provide the substrate to take neuromorphic computing as a tool for neuroscience to the next level without costly chip development.

The objectives of this thesis are:

- to introduce the concept of *model verification and substantiation* as a methodology for accumulating evidence of a model's plausibility and correctness, even in the absence of experimental validation data;
- to demonstrate a rigorous model verification and validation process, including an investigation of the required numerical precision on the SpiNNaker neuromorphic system;
- to develop and evaluate a novel FPGA-SoC-based neuromorphic compute node architecture capable of performing simulations in hyper-real-time, with the design strictly driven by neuroscience requirements, as a complementary yet distinct approach to the neuromorphic developments aiming at brain-inspired novel computer architectures for solving real-world tasks;
- to explore commercial off-the-shelf FPGA-SoC device technology and investigate its suitability as substrate for neuromorphic computing for application in computational neuroscience, in modeling and simulation; and
- to examine the technical requirements for large-scale neuromorphic computing and to propose a concept for a neuromorphic system integration that incorporates high-performance computing (HPC), taking into account the existing tools and workflows for modeling and simulation.

1.2 Spiking Neural Network Simulations in Neuroscience

This section introduces elementary notions of neurobiology and the basic concepts of numerical simulation of spiking neural networks in computational neuroscience. A comprehensive introduction to the complex fields is beyond the scope of this thesis. The presentation in this section is therefore highly selective, focusing on those aspects necessary for further understanding.

1.2.1 Spiking Neurons

Neurons are the primary functional units of the nervous system. They generate electrical signals that convey information. There are many different types of neurons, but most of them share the same main features. A typical neuron can be divided into three distinct parts: a cell body; dendrites; and an axon. These three parts are anatomically separate and serve different purposes. A schematic drawing of a neuron is shown in Figure 1.1A. The cell body (Greek soma) performs the essential life functions of a neuron by supporting all its chemical processes. Dendrites are projections from the soma, cellular extensions with many branches. Their function is to receive signals from other neurons or sensory information from the environment. The axon is a cable-like projection, the component of the neuron that conveys the electrical signals generated by the neuron and received by other neurons. These signals are all-or-none impulses, called *action potentials* or *spikes* (Figure 1.1B). Axons can vary greatly in length, ranging from 0.1 mm to 3 m (Kandel et al., 2000). Long axons are coated with an insulating sheath of myelin that is interrupted at regular intervals by the *nodes of Ranvier*. Action potentials traveling down the axon regenerate at these uninsulated spots, preventing signal degradation.

The branches of a neuron's axon form connections to many other neurons, the recipients of the signals. The point at which two neurons communicate is called a *synapse*. Accordingly, the neuron transmitting an action potential is called the *presynaptic neuron* and the neuron that is receiving this signal is called the *postsynaptic neuron*. Figure 1.1C shows a schematic drawing of a chemical synapse, which is the most common synapse type[1]. Axon branches end in what are called *presynaptic terminals*. They attach to the dendrites (and also the cell bodies) of the postsynaptic neurons, where they form synapses. At these contact points pre- and postsynaptic neuron are anatomically separated by a tiny space, the *synaptic cleft*.

A synapse translates electrical stimuli into chemical signals, i.e., the release of neurotransmitters from the presynaptic terminal into the synaptic cleft. The neurotransmitter molecules are detected by receptors in the postsynaptic cell membrane, where they open specific channels that initiate an ionic current flow into the cell. This current flow produces a so-called *postsynaptic potential* (PSP) and gradually changes the electrical charge of the neuron. A synapse here can be excitatory (the change in charge is positive) or inhibitory (the change in charge is negative), where a neuron forms either excitatory or inhibitory synaptic connections at all of its axonal branches. This is a consequence of *Dale's principle*, after Sir Henry Dale, a British physiologist who, in 1935, stated that a neuron performs identical chemical actions at all of its synaptic connections to other cells.

[1] Apart from chemical synapses, some operate purely electrical. These are called gap junctions.

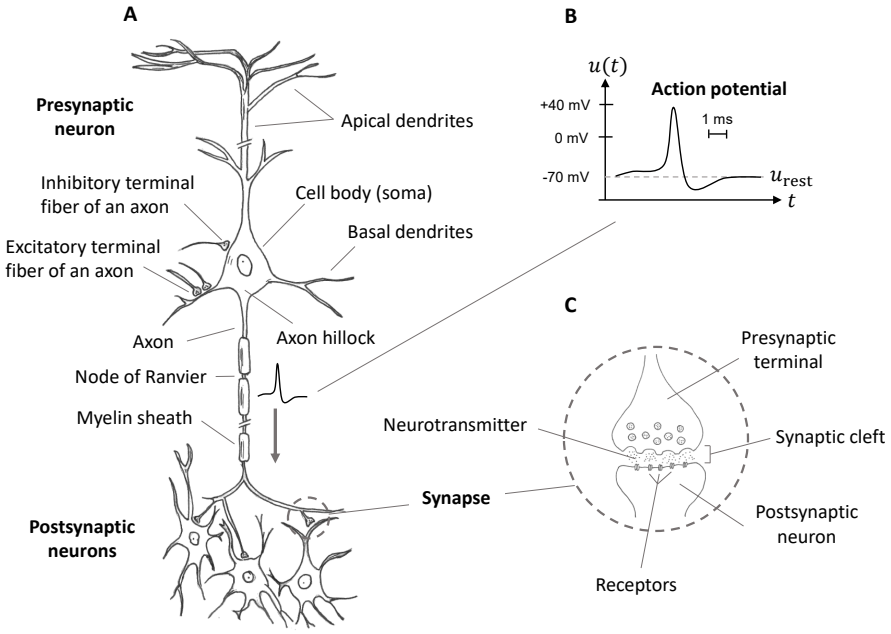


Figure 1.1 | Structure of a neuron. (**A**) Structure of a neuron schematically redrawn after Kandel et al. (2000) and modified. A neuron can be divided into three distinct parts: a cell body (soma), dendrites, and an axon (see main text for a description of their function). (**B**) Neurons generate stereotyped, uniform electrical signals, called *action potentials*. (**C**) The branches of a neuron's axon form communication sites with many other neurons. The point at which two neurons communicate is called a *synapse*. Shown here is a schematic drawing of a chemical synapse.

Synapses are dynamic plastic elements whose behavior is shaped by the previous history of both presynaptic and postsynaptic activity. Changes in synaptic signal transmission arise from a number of synaptic plasticity mechanisms, which gives synapses an active role in information processing (Abbott and Regehr, 2004).

A neuron maintains an electrochemical gradient between the inside and the outside of the cell, resulting in an electrical charge of the neuron. The synaptic inputs of a neuron continually change this charge. This can be measured by injecting an electrode into the neuron's cell body, recording the voltage across the cell membrane. The sketch in Figure 1.2 illustrates this. At its resting state, to which a neuron always tends to return, this membrane voltage, or *membrane potential*, has a value of approximately -70 mV (Dayan et al., 2005) (see also Figure 1.1B); the cell membrane here has a strong negative polarization. If the membrane potential exceeds some threshold value,

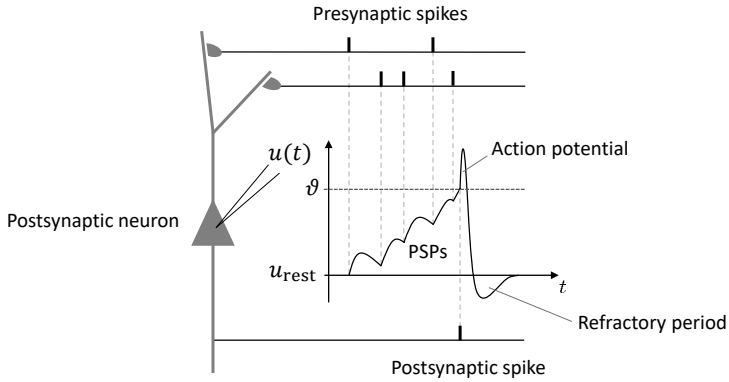


Figure 1.2 | Neuronal dynamics. The membrane potential $u(t)$ of a postsynaptic neuron changes depending on the synaptic input; here an excitatory input from two presynaptic neurons. Each incoming spike event evokes a postsynaptic potential (PSP), which decays over time. When the membrane potential exceeds some threshold value ϑ, the neuron generates an action potential; it emits a spike. After a refractory period, during which the neuron is unable to generate another action potential, the membrane potential returns to the resting state u_{rest}.

i.e., the synaptic input (the superposition of PSPs) is strong enough, the neuron generates an action potential; the neuron emits a *spike*. After emitting a spike, the neuron enters a refractory period during which it is unable to generate another action potential. One could also say that a neuron *integrates* incoming excitatory and inhibitory signals into a single cell response.

In the nervous system, billions of neurons work together forming complex communication networks. To study these networks, computational neuroscience uses mathematical models and numerical simulations.

1.2.2 Phenomenological Models

Depending on the scientific question, computational neuroscience employs different types of models with different levels of abstraction and complexity. Figure 1.3 compares three categories of models that use different abstraction: compartmental neuron models; point neuron models; and population-based models. Compartmental neuron models include the morphology of neurons, i.e., the spatial structure of dendritic trees and axons, which they decompose into many compartments. They can represent the electrophysiological properties of neurons with a high degree of accuracy, but due to their intrinsic complexity, their simulation comes at a very high computational cost. Point neuron models are not morphologically detailed. They are less complex and reduce a neuron to a single compartment, where the modeling of neuronal dynamics is grounded in

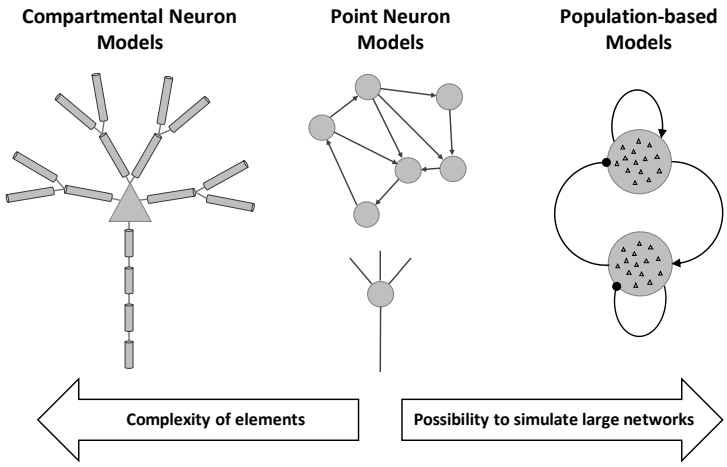


Figure 1.3 | Model classes with different levels of abstraction. In computational neuroscience, different types of models are employed. Here three levels of abstraction and complexity are shown. Compartmental neuron models are morphologically detailed. Correspondingly, these models are complex. Point neuron models reduce this complexity by modeling selected phenomenological properties. An even higher level of abstraction is used in population-based models, which describe the coarse-grained activity of large populations of neurons sharing the same properties. The computational cost of simulation increases with the complexity of a model, the amount of detail it contains. Reducing complexity, in turn, reduces computational cost, allowing the simulation of larger networks.

phenomenological properties. Point neuron models are suited for studies of network dynamics and memory. The reduction in complexity allows the simulation of larger spiking neural networks. A high level of abstraction is used in population-based models. These models describe the coarse-grained activity of populations of neurons sharing the same properties. Population-based models contain less detail, but allow for the simulation at a large scale.

The variety of models used in computational neuroscience is large (see, e.g., Gerstner et al., 2014; Dayan et al., 2005), and new models are being added continually. In this thesis, only point neuron models are considered. The neuromorphic compute node that is presented in Chapter 3 was developed with the objective of hyper-real-time simulation of networks of neurons belonging to this specific category of models.

In order to give an example of a basic point neuron model, the leaky integrate-and-fire (LIF) model is presented in the following. In addition, for a more thorough picture, a simple synapse model is also described, and some technical aspects of simulation are explained. Another point neuron model, the Izhikevich neuron model, will be described in Section 2.3.1.1.

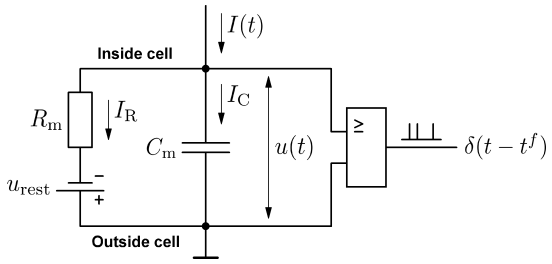


Figure 1.4 | Equivalent circuit diagram of the LIF neuron model. The capacitor C_m acts like the cell membrane. The voltage $u(t)$ across the capacitor is representing the membrane potential. In absence of an input current $I(t)$, the capacitor charges to the voltage u_{rest}. The *neuron* is then in the resting state, with the membrane potential $u(t) = u_{rest}$. A positive input current increases the charge of the capacitor. When $u(t)$ reaches the threshold ϑ, a spike event is generated. Spike events are represented as Dirac pulses $\delta(t - t^f)$, where t^f is the time at which a spike event occurs. If the driving current $I(t)$ vanishes, $u(t)$ returns to the resting value with the time constant $R_m C_m$, where R_m represents the neuron's membrane resistance.

Leaky integrate-and-fire (LIF) neuron model

The concept of the LIF model can be traced back to Louis Lapicque, a French physiologist, who, in 1907, published a study in which he introduces a model of electrical excitation of nerves that is based on a capacitor (Lapicque, 1907; Brunel and Van Rossum, 2007b,a). The LIF model is one of the simplest models and widely used in computational neuroscience.

The model describes neuronal dynamics as an integration process combined with a mechanism to trigger an action potential when the membrane voltage reaches some threshold. Action potentials are reduced to *events*, exploiting the fact that they always have roughly the same stereotyped shape. The model can be expressed as an RC-circuit. The circuit diagram is shown in Figure 1.4.

Subthreshold dynamics: The LIF model characterizes a neuron by a membrane voltage $u(t)$ that has a resting value u_{rest} (the resting potential of the neuron), a membrane capacitance C_m (the integrator), and a membrane resistance R_m (the leakage). Synaptic input is represented by an injected current $I(t)$. By applying Kirchhoff's law we can write $I(t)$ as

$$I(t) = I_C + I_R, \tag{1.1}$$

where I_C is the current that charges the capacitor C_m, and I_R is the current that passes through the resistor R_m. From Equation (1.1), using Ohm's law and the capacitor equation $I_C = C_m \frac{du}{dt}$, we derive

$$I(t) = C_m \frac{du}{dt} + \frac{u(t) - u_{rest}}{R_m}. \tag{1.2}$$

The standard form of the model (see, e.g., Gerstner et al., 2014) rewrites Equation (1.2) as

$$\tau_{\mathrm{m}} \frac{du}{dt} = u_{\mathrm{rest}} - u(t) + R_{\mathrm{m}} I(t), \tag{1.3}$$

where τ_{m} is interpreted as the membrane time constant with $\tau_{\mathrm{m}} = R_{\mathrm{m}} C_{\mathrm{m}}$, the time constant of the *leaky integrator*. The linear differential Equation (1.3) describes the evolution of the membrane potential $u(t)$; in electrical terms, it describes the voltage response of the RC-circuit to an input current.

Threshold and refractory: The generation of action potentials is modeled as an artificial process. Action potentials are expressed as events that happen at precise times t^f, that is, when the membrane potential $u(t)$ reaches the threshold value ϑ from below. This gives the threshold conditions for the emission of the fth ($f = 1, 2, \dots$) spike at time t^f as

$$t^f : \quad u(t^f) \geq \vartheta \quad \text{and} \quad t^f = \{t | u(t) \geq \vartheta\}. \tag{1.4}$$

When a spike event has occurred, the model enters a refractory period and is reset

$$\forall f, \ \forall t \in \left(t^f, t^f + \tau_{\mathrm{ref}}\right] : \quad u(t) = u_{\mathrm{reset}}. \tag{1.5}$$

In Equation (1.5), τ_{ref} denotes the absolute refractory time, and u_{reset} is the membrane reset potential, for which $u_{\mathrm{reset}} < \vartheta$ has to apply.

Spike events: A spike event is represented as an infinitely tall pulse in the form of the Dirac δ-function $\lim_{t \to t^f} \delta(t - t^f) = \infty$. A *spike train* can thus be denoted as a sequence of spike times

$$s(t) = \sum_f \delta(t - t^f). \tag{1.6}$$

The LIF model (Equation (1.3)) describes the voltage response to the time-dependent input current $I(t)$, the driving synaptic input. As an example of how this input is derived, the following describes a simple synapse model. This synapse model will use an alpha-function to shape synaptic inputs.

Current-based alpha-shaped synapse

Generally, synapse models can be divided into two basic types: conductance-based (COBA); and current-based (CUBA) models. The former respond to a presynaptic spike with a postsynaptic

potential (PSP), the latter with a postsynaptic current (PSC). The model presented here as an example is a current-based model frequently used in computational neuroscience.

The model describes the time course of a PSC by the alpha-function

$$\alpha(t) = \frac{t}{\tau_s} e^{1-t/\tau_s} \Theta(t), \tag{1.7}$$

which exponentially rises and decays with the time constant τ_s, the synaptic time constant. In Equation (1.7), Θ is the Heaviside step function with $\Theta(t) = 1$ for $t > 0$ and $\Theta(t) = 0$ else. The use of an alpha-function here accounts for the low-pass characteristics of synaptic transmission. The PSC that a spike event evokes in a postsynaptic neuron is derived by multiplying the alpha-function by a factor. A PSC is then given as

$$\mathrm{PSC}_{ij}(t) = w_{ij}\alpha(t), \tag{1.8}$$

where the factor w_{ij} determines the amplitude of the PSC, specifying the synaptic weight, i.e., the strength of the synaptic connection between the presynaptic neuron j and the postsynaptic neuron i. For a positive w_{ij}, the PSC is positive and the synapse is excitatory; a spike event induces an excitatory postsynaptic current (EPSC). If w_{ij} is negative, the PCS is negative and the synapse is inhibitory; a spike event results in an inhibitory postsynaptic current (IPSC). The superposition of PSCs gives the total synaptic input current of a neuron i, which can be formulated as

$$I_i(t) = \sum_j (\mathrm{PSC}_{ij} * s_j)(t - d_{ij}). \tag{1.9}$$

In Equation (1.9), $s_j(t)$ denotes the spike train of the presynaptic neuron j according to Equation (1.6), d_{ij} is the synaptic transmission delay from the presynaptic neuron j to the postsynaptic neuron i, which accounts for the dendritic and axonal delays of the connection, and "$*$" is the convolution operator, which is defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(s)g(t-s)\,ds. \tag{1.10}$$

The modeling of synapses has a technical aspect that requires explanation, as it is relevant for digital simulation and thus for the architecture of a neuromorphic accelerator.

Each synapse, i.e., each PSC-kernel, can in principle have a different time constant τ_s. Technically, in a simulation, this would require a neuron to maintain a PSC for each of its incoming

connections, resulting in an enormous computational cost and memory requirement when simulating a large network. Therefore, implementations typically use only two synaptic time constants per neuron, one for excitatory and one for inhibitory synapses, reducing the number of PSCs to be maintained to two. This is a generally accepted simplification, which allows a postsynaptic neuron to lump together the weighted spike trains of all excitatory presynaptic neurons, and lump together the weighted spike trains of all inhibitory presynaptic neurons. Thus, in a simulation, synaptic weights can be accumulated at spike arrival time and the combined effect of all synapses can then be incorporated into the model dynamics (see Rotter and Diesmann, 1999).

We can reformulate Equation (1.9) as two equations that divide the synaptic input current into an excitatory $\mathcal{E}$ component (the superposition of EPSCs)

$$I_{\mathcal{E},i}(t) = \alpha(t, \tau_s = \tau_{\mathcal{E}}) \sum_{j \in \mathcal{E}} (w_{ij} * s_j)(t - d_{ij}), \tag{1.11}$$

and an inhibitory $\mathcal{I}$ component (the superposition of IPSCs)

$$I_{\mathcal{I},i}(t) = \alpha(t, \tau_s = \tau_{\mathcal{I}}) \sum_{j \in \mathcal{I}} (w_{ij} * s_j)(t - d_{ij}), \tag{1.12}$$

where τ_{E} and τ_{I} denote the synaptic time constants of excitatory and inhibitory synapses, respectively. The total synaptic input current of a neuron i is then given by

$$I_i(t) = I_{\mathcal{E},i}(t) + I_{\mathcal{I},i}(t). \tag{1.13}$$

LIF models with current-based synapses have an advantage in digital simulations. They allow exact integration (see, e.g., Rotter and Diesmann, 1999; Morrison et al., 2007). However, this is generally not the case for the majority of models. The dynamical systems that the mathematical models of neurons and synapses describe are typically nonlinear and require numerical methods for their solution.

Mathematical models of neurons and synapses are the basis for the construction of network models, which arrange neurons into populations, and connect populations to cortical circuits, and cortical circuits to brain areas, and so forth. These models can be supplemented with further phenomenological properties, for instance the description of topologies and rules for plasticity (a spike-timing-dependent plasticity (STDP) rule is described in Section 2.3.1.1). Due to their nonlinear and complex nature, digital simulation is the tool of choice to study these systems.

1.2.3 Digital Simulation

Digital simulation plays a key role in gaining insight into the underlying principles of neural computation. It enables neuroscientists to study the interaction of neurons and synapses and the complex dynamics that arise when large numbers of them are connected into networks.

When simulating a dynamical system on a digital computer, we generally have to differentiate between a *discrete-time* simulation and a *discrete-event* simulation. In a discrete-time simulation, the time axis is divided into evenly spaced intervals, a time grid. A continuously evolving system is then advanced in steps using discrete-time approximation methods – state changes are considered at specific points in time and not continuously through time. In contrast, in a discrete-event simulation, the system is advanced when an event occurs, where an event is an action that affects the state of the system. The two simulation paradigms are also known as *time-driven* and *event-driven* simulation. Both have their advantages and disadvantages. However, the pros and cons will not be the subject of further discussion here, as they are of secondary interest.

Software tools for the simulation of spiking neural networks, such as the neural simulation tool NEST (Gewaltig and Diesmann, 2007), typically use a hybrid simulation scheme. Neuron states are updated in a time-driven manner, whereas spike events are processed in an event-driven way (Morrison et al., 2005). This is a reasonable choice: the time-driven scheme allows the network state to be advanced effectively, and the event-driven scheme is appropriate because between the occurrence of two successive spike events arriving at a neuron, the weighted synaptic input to the neuron does not change – the process is inherently event-discrete. The flow diagram in Figure 1.5 illustrates this hybrid simulation scheme.

Notions of time

The system is evaluated in steps where the time axis is divided into intervals

$$t_k \leq t \leq t_{k+1}, \quad t_{k+1} = t_k + h. \tag{1.14}$$

In Equation (1.14), h is the temporal spacing of the grid, i.e., the time resolution of the simulation. The indices k enumerate the simulation time steps. The simulated time is then given by

$$T = kh. \tag{1.15}$$

The term *simulated time* refers to the biological time domain and is used here to avoid ambiguities with the term *simulation time*. The latter is also often used to refer to the duration of a simulation,

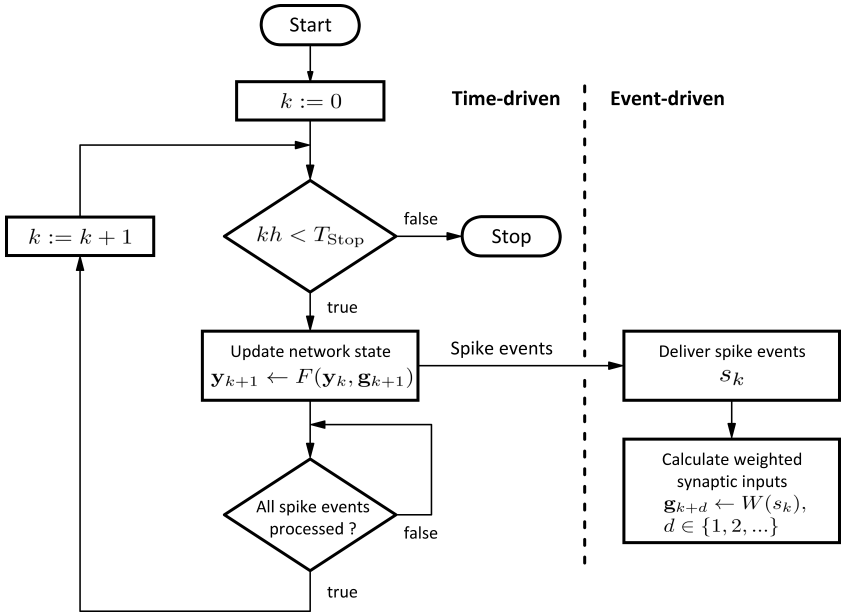


Figure 1.5 | Hybrid simulation scheme. The network state is advanced at evenly spaced time intervals, i.e., time-driven, whereas spike events are processed as they occur, i.e., event-driven. See main text for description.

i.e., the physical time elapsed in the system.

State update

In each simulation time step, each neuron is *visited* and its dynamic behavior is approximated on the time grid by employing numerical methods. Depending on the type of ordinary differential equations (ODEs) that describe these dynamics, Euler or Runge-Kutta-Fehlberg methods are often used here. A state transition from one grid point to the next is based on the current state $\boldsymbol{y}_k$ and incorporates the weighted synaptic inputs $\boldsymbol{g}_{k+1}$, the summed synaptic weights of all events arriving at time step $k+1$: $\boldsymbol{y}_{k+1} \leftarrow F(\boldsymbol{y}_k, \boldsymbol{g}_{k+1})$. At the grid points, the system is always in a well-defined state.

Spike events, s_k, are processed as they occur, calculating the synaptic inputs that will be incorporated into the dynamics at subsequent time steps: $\boldsymbol{g}_{k+d} \leftarrow W(s_k)$, $d \in \{1, 2, ...\}$. To preserve the temporal causality of events, the time-driven process may need to wait until the events that occurred during a state transition are processes.

This hybrid simulation scheme has proven efficient and is also used by the neuromorphic compute node presented in Chapter 3.

1.2.4 Methods and Tools

In the process of gaining insight into the underlying principles of neural computation, the methods and software tools developed and provided by the computational neuroscience community play a key role. The tools listed below are only a selection with a focus on modeling and simulation.

To conveniently describe the dynamics of neuron and synapse models, domain-specific languages (DSLs) such as NeuroML (Gleeson et al., 2010), NMODL (Hines and Carnevale, 2000), and NESTML (Plotnikov et al., 2016) have been developed. These tools allow models to be formulated in a high-level language, a DSL description, from which tools then generate simulation code.

Community software for simulation has been developed to run on all scales from laptops to the largest supercomputers. Examples are the simulation engines NEURON (Hines and Carnevale, 1997), Arbor (Akar et al., 2019), NEST (Gewaltig and Diesmann, 2007), and Brian (Goodman and Brette, 2008). For convenient graphical user interaction, tools such as NEST Desktop (Spreizer et al., 2021) are also available.

Network models are often described using a higher-level language such as PyNEST (Eppler et al., 2009) or PyNN (Davison et al., 2009). PyNN, for example, provides a common user interface for software-based neural network simulation tools (e.g., NEURON, NEST, and Brian) and also supports the neuromorphic systems SpiNNaker (Furber et al., 2013) and BrainScaleS (Schemmel et al., 2010). Here the Python programming language has established as front-end for user interaction. The language is easy to learn, has a large active community base, and offers extensive support for numerical calculations and data analytics.

This software landscape is complemented by numerical tools for statistical analysis, such as the Electrophysiology Analysis Toolkit *Elephant*[2] as well as tool support for model validation methodologies, for example, the validation framework *NetworkUnit*[3] (Gutzen et al., 2018).

The requirements with respect to efficiency, correctness and the reproducibility of results place high demands on these tools and the entire software ecosystem. This infrastructure and its requirements must be taken into account when developing new tools or designing a novel neuromorphic system if it should be of value for the computational neuroscience community.

[2] RRID:SCR_003833; http://neuralensemble.org/elephant
[3] RRID:SCR_016543; https://github.com/INM-6/NetworkUnit

1.3 Neuromorphic Computing as Tool for Neuroscience

The main driving force behind the developments in neuromorphic computing is undoubtedly brain-inspired computing, which seeks to apply new technologies and develop novel computer architectures inspired by the working principles of the brain – although these are not yet fully understood. The application of neuromorphic computing in computational neuroscience, however, is still a niche area, but of high interest in the field. As a tool to accelerate simulations, it can help unlock the secrets of brain function and thus contribute to the theoretical foundations of practical applications.

When studying neural networks, it is generally desirable to simulate them as fast as possible. Whereas real-time simulation is interesting because of the possibility of interacting with real-world applications, hyper-real-time would enable the study of slow processes such as learning and memory, and permit researchers to perform more comprehensive parameter scans of faster processes. However, not even the fastest and most advanced supercomputers available today can meet the challenge of significantly accelerating the simulation of a large-scale spiking neural network. The use of neuromorphic computing for this purpose, leveraging novel technologies and application-specific hardware architectures, is therefore highly attractive as it promises to provide the necessary tools for this task.

1.3.1 Requirements

Despite all technological innovations, making neuromorphic computing a useful tool in computational neuroscience modeling and simulation is a demanding technical challenge. Neuroscience research employs mathematical models to gain understanding of the complex dynamics of neural networks. Their simulation requires numerical accuracy. Simulations must be deterministic with reproducible outcomes. The plethora of neuron and synapse models used in simulations makes it difficult to arrive at an architecture design that satisfies all requirements equally. Plasticity rules and algorithms, which is a rapidly evolving area of research, also require a high degree of flexibility in algorithmic implementation.

Moreover, there are questions that lack a clear answer, leaving design decisions in a state of uncertainty. One such question is, for example, the required numerical precision. The answer here determines the specification of data types and the implementation of arithmetic operations and algorithms – design decisions that affect implementation complexity, chip area, and power efficiency. To the best of my knowledge, so far, only a few studies have examined the effects of numerical accuracy on simulation outcomes (e.g., Pfeil et al., 2012; Gutzen et al., 2018; Dasbach

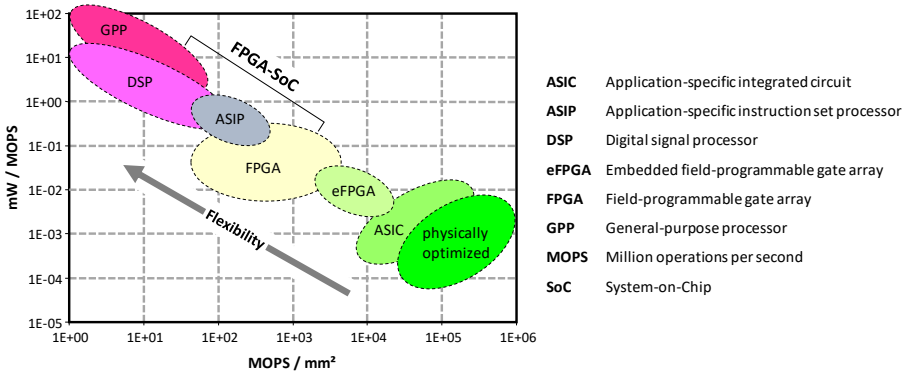


Figure 1.6 | Energy vs. flexibility conflict. Invers energy efficiency and area efficiency of different implementation styles (scaled to a 130 nm CMOS technology) redrawn after Noll et al. (2010) and modified. The alternatives span more than five orders of magnitude in both energy and area efficiency. Flexibility increases toward general-purpose solutions as efficiency decreases. From an application perspective, FPGA-SoC technology can bridge the flexibility-gap between programmable logic devices and general-purpose processors.

et al., 2021; Trensch et al., 2018, and Chapter 2).

A neuromorphic architecture must therefore provide flexibility in model implementation. It should not be bound to a specific model and should be open to extensions. To achieve high user acceptance, it must integrate with the existing landscape of tools and workflows for modeling and simulation. Scalability and the capability of delivering significant acceleration are of course also key requirements.

Flexibility and efficiency are both essential, yet they are technically conflicting requirements that constrain the design space and influence the choice of technology and implementation style. The requirement for numerical accuracy and reproducibility here rules out approaches based on analog circuits.

1.3.2 Choice of Technology

Flexibility and efficiency are opposing goals in the choice of technology, resulting from the so called *energy vs. flexibility conflict* (Noll et al., 2010). This conflict is illustrated in Figure 1.6, which shows a quantitative comparison of different circuit implementation styles.

While traditional programmable general-purpose processors (GPPs) provide the highest level of flexibility, a physically optimized application-specific integrated circuit (ASIC) *freezes* a specific use case in silicon. Physically optimized ASICs achieve the highest efficiency, but are inflexible.

Between technologies, energy and area efficiencies here span more than five orders of magnitude. From Figure 1.6, it can be seen that field-programmable gate arrays (FPGAs) can provide a good compromise between flexibility and efficiency.

FPGAs have long been the exclusive domain of hardware engineers, primarily used in rapid prototyping. Software developers often didn't even know what they were. This has changed, not least because of advances in the EDA[4] tools, which today allow the designer to describe a hardware design at a higher level of abstraction, for example, in the C language as an algorithmic description. These High-Level Synthesis (HLS) methods reduced development times and made the technology accessible to non-hardware experts. This has also set impulses for neuromorphic computing and led to a number of FPGA-based developments in the field (see Section 1.3.3).

Today's broader use of FPGAs has also been facilitated by FPGAs empowered with general-purpose processors, which has also changed the way FPGAs are used. This type of devices emerged a decade ago. The latest generation of this technology integrates a programmable logic device together with a complete computer system and large memories in a single chip, a System-on-Chip (SoC). Since its emergence, commercial off-the-shelf FPGA-SoC technology has gained popularity, not least because of its ability to combine the flexibility of a general-purpose processor with the efficiency of application-specific logic. From an application perspective, this technology mitigates the *flexibility vs. efficiency conflict* (in Figure 1.6 indicated by the overarching label *FPGA-SoC*) and enables novel approaches for architecture designs.

It is specifically these characteristics that make FPGA-SoC devices an interesting substrate for neuromorphic computing for application in computational neuroscience. The suitability of the technology for this purpose is the subject of further investigation in this thesis.

1.3.3 Neuromorphic Developments

Neuromorphic computing already has a rich history. The field has received broader attention in recent years, largely because of its potential to enable low-power machine intelligence and edge applications that require real-time data processing. The most prominent efforts in this regard are Intel's Loihi platform (Davies et al., 2018) and the IBM NorthPole neural inference machine (Modha et al., 2023a,b), which is IBM's latest development and the successor to the TrueNorth chip (Merolla et al., 2014). A complete overview of the field would exceed the scope of this thesis. Therefore, I will only highlight some developments with a focus on neuroscience simulation and emphasize aspects that are relevant here. Surveys into to the field can be found in, e.g., Schuman et al. (2017, 2022).

[4] Electronic Design Automation

BrainScaleS (Heidelberg University) **SpiNNaker** (University of Manchester)

Figure 1.7 | Large-scale neuromorphic systems developed in the Human Brain Project.

Developed to a large extent within the Human Brain Project[5] and dedicated to neuroscience are the large-scale neuromorphic systems BrainScaleS[6] (Schemmel et al., 2010) and SpiNNaker[7] (Furber et al., 2013). The systems are shown in Figure 1.7. They are based on two complementary principles.

The BrainScaleS system and its successor BrainScales-2 (Pehle et al., 2022), which both have been developed at Heidelberg University, are capable of running simulations three orders of magnitude faster than real-time. To achieve this, the architecture builds on the physical, i.e., analog, emulation of neuron and synapse models (Schemmel et al., 2017) in dedicated mixed-signal circuits. This is combined with digital plasticity processors (only BrainsScaleS-2; see Friedmann et al. (2017)). The system uses wafer-scale integration to efficiently interconnect a large number of analog neurons to accommodate the high speed-up. A single wafer incorporates about $2 \cdot 10^5$ neurons and $44 \cdot 10^6$ synapses. In the left of Figure 1.7, a 20-wafer machine is shown that is located at Heidelberg University. Physical, analog emulation restricts the system to its built-in, *silicon-frozen* analog models, and use cases where technology-related effects, such as fabrication tolerances and thermal noise, are acceptable. The very high speed-up, however, makes it highly suitable to applications that take a very long time in the biological domain.

The SpiNNaker system was developed at the University of Manchester. It is a massively parallel neuromorphic computing platform based on energy-efficient digital multi-core ARM chips. It is fully programmable, allowing flexibility in the choice and implementation of numerical models, and enables large-scale simulations to be performed in real time. The current largest

[5] The Human Brain Project (HBP) (2013 – 2023) was a European flagship project that pioneered digital brain research.
[6] BrainScaleS is a contraction of *brain-inspired multiscale computation in neuromorphic hybrid systems*.
[7] SpiNNaker is a contraction of *spiking neural network architecture*.

SpiNNaker system is a one million core machine located in Manchester (shown in the right of Figure 1.7). What sets SpiNNaker apart and differentiates it from a *normal* supercomputer is its brain-inspired communication fabric, which is optimized for broadcasting large numbers of small data packets, ideal for conveying spike events. The successor SpiNNaker2 (Höppner et al., 2021) is developed at TU Dresden. SpiNNaker2 is an evolution of the first generation SpiNNaker architecture, adding dedicated accelerators and generally increasing system performance.

BrainScaleS and SpiNNaker are milestones behind which are enormous development efforts, not least because of time-intensive and costly chip developments. Both are integrated into the landscape of tools and workflows for neuroscience simulation and are accessible through the European research infrastructure EBRAINS[8].

As substrate for neuromorphic computing, FPGAs are becoming increasingly attractive. With today's FPGAs, complex designs can be realized quickly and costly chip development can be avoided. In an early study, Maguire et al. (2007) made an inventory and revealed the challenges associated with implementing large-scale spiking neural networks using FPGAs. A number of architectural approaches and implementations for different use cases have since been published. Below, I just highlight a few selected developments, looking at them from the perspective of neuroscience simulation.

Bluehive (Moore et al., 2012) is a scalable custom 64-FPGA machine that is dedicated to the simulation of large-scale networks with demanding communication requirements. On a single FPGA, *Bluehive* can simulate 64,000 Izhikevich neurons in real time. *NeuroFlow* (Cheung et al., 2016) is a platform that builds on top of Maxeler's[9] Dataflow Engine (DFE) technology. A 6-FPGA system can simulate networks consisting of 600,000 neurons. Real-time performance is achieved when simulating 400,000 neurons. The simulation of a plastic 1000 neuron two-population Izhikevich model for 24 h biological time can be completed in 1435 s, thus achieving an approximately 60-fold acceleration. The platform supports several neuron and synapse model types and a spike-timing-dependent plasticity (STDP) rule. *NeuroFlow* also provides a PyNN interface. In Wang et al. (2014) and Wang et al. (2018) an architecture is proposed that uses a procedural *on-the-fly* generation scheme for parameters and connections and can simulate 20 million to 2.6 billion LIF neurons in real time on a single Stratix V FPGA.

In the designs of the above systems, engineers had to make trade-offs with respect to simulation accuracy and model complexity to achieve the desired system size and performance. For example, all the above systems advance neuron dynamics in steps of 1.0 ms; this update interval is ten times larger than the *de facto* standard used in digital simulations. This significantly reduces

[8] https://www.ebrains.eu/
[9] Maxeler Technologies: www.maxeler.com

computational cost, but also reduces numerical accuracy – especially for neuron models with stiff equations (Blundell et al., 2018c; Hansel et al., 1998; Morrison et al., 2007; Pauli et al., 2018). Model complexity was reduced in the architecture proposed in Wang et al. (2014) and Wang et al. (2018); individual synaptic connection delays are replaced by an axonal delay. This avoids large memory structures and the computational cost of delaying and accumulating incoming synaptic events, allowing millions of neurons to be simulated on a single FPGA. However, the limitations accepted here may well be appropriate for a variety of applications.

FPGA-based systems that aim for accurate and reproducible simulation typically show more limited system characteristics in terms of the number of neurons per node (FPGA device) simulated and the achievable acceleration factor. For example, in Pani et al. (2017) a scalable modular architecture for closed-loop experiments with in vitro cultures is presented. The platform can simulate small to medium-sized networks in real time and implements (only) 1440 Izhikevich neurons.

Based on FPGA-SoC technology is the IBM Neural Computer prototype INC-3000 (Narayanan et al., 2020). A single-cage system clusters 432 AMD Xilinx Zynq SoC devices in a high bandwidth 3D mesh communication network. The system is highly flexible and applications can offload algorithms to programmable logic to accelerate them. In Heittmann et al. (2022), the INC-3000 system was employed for a simulation of the cortical microcircuit model (Potjans and Diesmann, 2014), which comprises approximately $0.8 \cdot 10^5$ neurons and $0.3 \cdot 10^9$ synaptic connections (the model is described in more detail in Section 6.3.2). The simulation was implemented using High-Level Synthesis (HLS) and required 305 SoC devices on the INC-3000 system, where only the programmable logic part was used. The simulation achieved a speed-up factor of four compared to the biological time domain. This speed-up factor is mainly determined (limited) by the spike exchange times and hop latencies in the INC-3000 systems' communication network, which is optimized for bandwidth but not for latency (see Heittmann et al. (2022)). However, at the time of publication, it was the fastest simulation of this model. In Kauth et al. (2023), an FPGA cluster that connects 35 NetFPGA SUME boards in a two-hop low-latency communication network was used to implement a simulation of the same model. This simulation achieved an acceleration factor of 20. For both microcircuit model implementations, the correctness of the simulation outcome was validated by the authors against reliable reference data, which sets these implementations apart from others.

A system that can offer the flexibility of SpiNNaker while significantly accelerating the simulation of large-scale spiking neural networks would be a valuable next-generation platform for neuroscience research. The above developments point in this direction. However, no such system

exists yet, to the best of my knowledge.

1.4 Contributions

The contributions of this thesis are of methodological, technical, and conceptual nature.

For the field of neural network modeling and simulation, a reasonable adaptation of the existing terminology for model verification and validation is proposed that is more explicit and better expresses the underlying intent in the field. The concept of *model verification and substantiation* is introduced as a methodology that allows for the accumulation of evidence of a model's plausibility and correctness, even in the absence of experimental validation data. The methodology is demonstrated by conducting a rigorous model verification and validation process, where the issue of required numerical precision is investigated on the SpiNNaker neuromorphic system. It is shown that the appropriate choice of the numerical precision is critical for sufficient accuracy in reproducing model dynamics; even small deviations in the dynamics of individual neurons can affect the dynamics at the network level.

The main contribution of this thesis is the development and evaluation of a novel FPGA-SoC-based hybrid hardware and software mixed architecture for a neuromorphic compute node that is capable to perform simulations in hyper-real time. The design of the compute node is strictly driven by neuroscience requirements. This distinguishes the development from other efforts in the field, and presents a complementary yet distinct approach to the neuromorphic developments that aim at brain-inspired novel computing architectures for solving real-world tasks. Commercial off-the-shelf FPGA-SoC technology is explored for its suitability as substrate for neuromorphic computing for application in computational neuroscience modeling and simulation, where the technology is found to be suitable and holding great potential.

For the systematic assessment of the performance characteristics of the neuromorphic compute node, a workload model and a performance model are developed, introducing a metric that is independent of the size of a network and describing the performance-determining aspects of a hybrid time- and event-driven simulation scheme.

The technical requirements for large-scale neuromorphic computing for accelerated simulation are examined, and a basic concept for a neuromorphic system integration is proposed that incorporates the existing HPC landscape.

1.5 Thesis Outline

This thesis is structured as follows. Chapter 2 presents a rigorous model substantiation methodology for increasing the correctness of simulation results in the absence of experimental validation data. For the field of computational neuroscience a reasonable adaptation of the existing terminology for model verification and validation is proposed, and the concept of *model verification and substantiation* is introduced. The usefulness of the method, and the application of a systematic approach in general, is demonstrated by means of a worked example on the SpiNNaker neuromorphic system, showing that the appropriate choice of numerical precision and algorithms is critical for reproducing model dynamics.

In the main part of this thesis, in Chapter 3, a novel FPGA-SoC-based hybrid hardware and software mixed neuromorphic compute node architecture is presented that is capable to perform simulations in hyper-real time, where the design of the compute node is strictly driven by neuroscience requirements. The chapter first describes the setup of the development platform used for the prototypical implementation and gives a brief introduction to the FPGA-SoC device technology. This is followed by remarks on the chosen logic design methodology and the development workflow conducted. The main section of the chapter describes the architecture of the neuromorphic compute node, divided into a software architecture part and a hardware architecture part. The underlying principles and conceptual ideas are explained, and the function and microarchitecture of the main building blocks are described. Where of interest and relevant to the design, architecture alternatives are evaluated and discussed. Finally, the operating latencies, a breakdown of the utilized chip resources, and an estimate of the power consumption are presented.

In Chapter 4, the methods proposed in Chapter 2 are applied for a proof of correctness of the implementation and design of the neuromorphic compute node. The chapter begins by reviewing the design decisions made with respect to the required numerical precision and describes a calculation verification task that was conducted to evaluate the accuracy in reproducing the dynamics of individual neurons. This is followed by a description of the co-verification process that accompanied hardware-software co-development, and which employed a rather unusual hardware verification approach, referred to here as *in-FPGA verification*. This verification approach exploits the FPGA-SoC device technology to implement a software-driven testbench in the C language. The technique is briefly explained and compared to the implementation verification using behavioral simulation at the microarchitecture level. The setup of the testbench is described and an overview of the hierarchy of hardware-software function tests is given. Finally, the correctness of the implementation is demonstrated by conducting a substantiation assessment that compares the dynamics of a selected network state of a simulated test network model with

two reliable references.

The Chapter 5 presents a systematic assessment of the performance characteristics of the neuromorphic compute node. To this end, a workload model is developed that introduces a metric that is independent of the size of a network. This is followed by the presentation of a precise performance model derived from the microarchitecture, but capturing in a general way the performance-determining aspects of a hybrid time- and event-driven simulation scheme. The workload and performance model are then used to evaluate and predict the behavior of the compute node under varying workload situations and for different configurations and assumptions in the design space. Described are the assessment of the single-node performance and the prediction of the performance characteristics for cluster operation.

Looking ahead to a large-scale system, Chapter 6 proposes a number of modifications and extensions to the architecture presented in Chapter 3, and examines the technical requirements resulting from the higher workloads generated by large-scale networks. The chapter begins by reflecting on design decisions made for the neuromorphic compute node architecture with respect to on-chip memories. Additional parallelization options for spike processing and alternative architectures using memory partitioning are proposed for performance-critical components, enabling designs that can cope with high workloads. To derive a definition of these workloads, two large-scale neural network models widely used in neuroscience are analyzed. Based on this analysis, estimates for technical requirements and achievable performance are given. Finally, a basic architectural concept for an integration of a large-scale neuromorphic computing system into the HPC landscape is presented.

The closing chapter, Chapter 7, summarizes the discussions that conclude each of the chapters and provides an outlook on future developments.

Chapter 2

Rigorous Neural Network Simulations

A Model Substantiation Methodology for Increasing the Correctness of Simulation Results in the Absence of Experimental Validation Data

"An expert is someone who knows
some of the worst mistakes that
can be made in his subject,
and how to avoid them."
Werner Heisenberg

2.1 Introduction

The reproduction and replication of scientific results is an indispensable aspect of good scientific practice, enabling previous studies to be built upon and increasing our level of confidence in them. However, reproducibility and replicability are not enough: an incorrect result will be accurately reproduced if the same incorrect methods are used.

In computational neuroscience, the setup of a neural network simulation can be complex, and an incorrect result can have many possible causes. These range from inappropriate model implementations and data analysis methods, to procedural errors such as weaknesses in simulation planning, setup, and execution, to errors caused by hardware limitations such as insufficient numerical precision. Even for domain experts, it is therefore often difficult to judge the correctness of a result obtained from a complex neural network simulation. Moreover, some of the factors that affect correctness are beyond the control of the modeler or experimenter.

Credibility can be built by formalizing processes, i.e., following a systematic approach. This applies to the modeling, implementation, and simulation tasks performed in a particular experiment or study, as well as to their reproduction in a different setting. Correctness is understood here not as an absolute state of being correct or incorrect, but as a gradual process of refinement, progressively moving closer to a fully accurate result. Although appropriate methods, such as verification and validation methodologies, exist, they are not yet well established in the field of neural network modeling and simulation. One reason for this may lie in the rapid pace of the developments in the field, which impedes the development of common verification and validation methods; another is likely to be that the field has yet to absorb knowledge of these methodologies from fields where they are common practice. This latter aspect is exacerbated by partly contradicting terminology around these areas.

The first part of this chapter therefore addresses terminology and methodology. A reasonable adaptation of the existing terminology for model verification and validation is proposed and applied to the field of neural network modeling and simulation. The concept of *model verification and substantiation* is introduced as a methodology that allows for the accumulation of evidence of a model's plausibility and correctness, even in the absence of experimental validation data. In the second part of this chapter, this methodology is then applied to the issue of reproducibility, demonstrated by a worked example. Specifically, a minimal spiking network model capable of exhibiting the development of polychronous groups[1], as described in Izhikevich (2006), is quantitatively compared to its reproduction on the SpiNNaker neuromorphic system (Furber et al.,

[1] A polychronous group is characterized by a reproducible time-locked, spike-timing pattern over a group of neurons Izhikevich (2006).

2013). The Izhikevich (2006) study is highly cited as an account of how spike patterns emerge from network dynamics, and contains a number of non-standard features in its conceptual and implementational choices that make it a particularly illustrative example for the demonstration of a rigorous verification and validation process. The choice of the SpiNNaker neuromorphic system as a target for network reproduction is motivated by the fact that SpiNNaker is subject to rather different constraints from typical simulation platforms. In particular, the restriction to fixed-point arithmetic allows the demonstration of different verification problems. By means of a worked example, an iterative verification and validation process is outlined that demonstrates the usefulness of a systematic approach and the value of standard software engineering practices.

The insights gained from this process also guided design decisions in the development of the hybrid neuromorphic compute (HNC) node architecture that will be presented in Chapter 3. In particular, the results derived from calculation verification tasks, which examined the appropriateness of the choice of numerical precision and algorithms for calculating the dynamics of a test case model, influenced design decisions. In addition, the methods proposed in this chapter have also been used in an adapted form to prove the correctness of the implementation of the HNC node (see Chapter 4 for further details).

Contributions

- A reasonable adaptation of the existing terminology for model verification and validation is proposed for the field of computational neuroscience modeling and simulation.
- The concept of *model verification and substantiation* is introduced as a methodology that allows for the accumulation of evidence of a model's plausibility and correctness, even in the absence of experimental validation data.
- A rigorous verification and validation process is described by means of a worked example, demonstrating the usefulness of a systematic approach and the value of standard software engineering practices.
- As a result of a conducted calculation verification task, it is shown that for the selected model numerical precision is critical for sufficient accuracy in reproducing model dynamics; even small deviations in the dynamics of individual neurons can affect the dynamics at the network level.

2.2 Terminology

2.2.1 Reproducibility and Replicability

Reproducibility and replicability are indispensable aspects of good scientific practice. Unfortunately, the terms are defined in incompatible ways across and even within fields.

In psychology, for example, *reproducibility* may mean completely re-doing an experiment, whereas *replicability* refers to independent studies that yield similar results (Patil et al., 2016). For computational experiments, where a deterministic outcome is typically expected[2], *reproducibility* is understood as obtaining the same results by a different experimental setup conducted by a different team (Association for Computing Machinery, 2016). Although attempts were made to help resolve the ambiguity in the terminology by explicitly labeling the terms or by attempting to inventory the terminology across disciplines (Barba, 2018), the problem persists. In Plesser (2018) a brief history of this confusion is given.

In this thesis, the definitions proposed by the *Association for Computing Machinery* is followed (Association for Computing Machinery, 2016):

> **Replicability** *(Different team, same experimental setup) "The measurement can be obtained with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using the author's own artifacts."*
>
> **Reproducibility** *(Different team, different experimental setup) "The measurement can be obtained with stated precision by a different team, a different measuring system, in a different location on multiple trials. For computational experiments, this means that an independent group can obtain the same result using artifacts which they develop completely independently."*

To be more specific about the terminology of reproducibility, in this work it is aimed for *results reproducibility* (Goodman et al., 2016; see also Plesser, 2018).

> **Results reproducibility** *"Obtaining the same results from the conduct of an independent study whose procedures are as closely matched to the original experiment as possible."*

[2] A particular input will always result in the same output. However, this is not always guaranteed, for example in analog neuromorphic computing. Here, the outcome is not only determined by the initial conditions. Chip fabrication tolerances and thermal noise add a stochastic component.

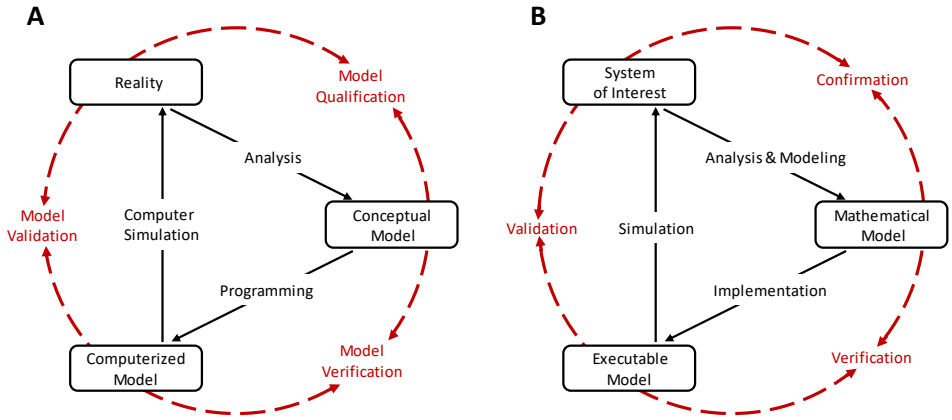


Figure 2.1 | Interrelationship of the basic elements for modeling and simulation. (A) Terminology for credibility of a modeling and simulation process as introduced by Schlesinger et al. (1979), depicting the *framework* with the basic elements and their interrelationships (figure redrawn). **(B)** Proposed terminology for modeling and simulation in computational neuroscience, for which a less generic terminology is more expedient. The adaptation of terms is inspired by the terminology used by Thacker et al. (2004). While the authors there use the terms *reality of interest*, *mathematical model*, and *computer model*, here the terms *system of interest*, *mathematical model*, and *executable model* are preferred as they better express the underlying intent in the field. The framework distinguishes between modeling and simulation activities (black solid arrows), and assessment activities (red dashed arrows).

2.2.2 Model Verification and Validation

The critical question for all modeling tasks is whether the model provides a sufficiently accurate representation of the system being studied. Evaluating the results of a modeling effort is a non-trivial exercise that requires a rigorous validation process.

The term *validation*, or more generally *verification and validation* also require a precise definition, as they have different meanings in different contexts. In software engineering, for example, verification and validation is the objective assessment of products and processes throughout the life cycle. Its purpose is to help the development organization build quality into the system (Bourque and Fairley, 2014). With respect to the development of computerized models, verification and validation are processes that accumulate evidence of a model's correctness or accuracy for a specific scenario (Thacker et al., 2004).

As a cornerstone for establishing credibility of computer simulations, the Society for Computer Simulation (SCS) formulated a standard set of terminology intended to facilitate effective communication between model builders and model users (Schlesinger et al., 1979). This early definition is very general and often does not do justice to a particular modeling domain. Therefore,

domain specific adaptations to the terminology can be found, but having fundamentally the same meanings. For the field of neural network modeling and simulation the proposed terminology is shown in Figure 2.1B, amended from Thacker et al. (2004). While Thacker et al. (2004) uses the terms *reality of interest*, *conceptual model*, and *computerized model*, the terms *system of interest*, *mathematical model*, and *executable model* are here proposed instead. The terms are more explicit and better express the underlying intent. In particular, due to the empirical challenges of neurobiology, spiking neural network models are often not based on a specific biological network that could be considered *reality* and from which ground truth behavior can be recorded, in contrast to, for example, the air flow around a wing. The term *system of interest* recognizes that the process of verification and validation can also be applied to systems without concrete physical counterparts.

The essence of the introduced terminology is the division of the modeling process into three major elements as illustrated in Figure 2.1A and Figure 2.1B.

> Reality or **system of interest** is an *"entity, situation, or system which has been selected for analysis"*. The conceptual or **mathematical model** is defined as a *"verbal description, equations, governing relationships, or natural laws that purport to describe reality or the system of interest"* and can be understood as the precise description of the modeler's intention (Schlesinger et al., 1979). The formulation of the conceptual or mathematical model is derived in a process called **analysis and modeling** and its applicability is motivated in a process termed qualification or **confirmation**. However, the conceptual or mathematical model by itself is not able to simulate the system of interest. By means of applying engineering and development effort it has to be implemented as an computerized or **executable model**.

By separating the understanding of a model into a mathematical model and an executable model, this terminology also expresses the difference between verification and validation.

> **Verification** describes the process of ensuring that the mathematical model is appropriately represented by the executable model, and improving this fit.

Model verification is the assessment of a model implementation. Neural network models are mathematical models that are written down in source code as numerical algorithms. Therefore, it is useful to define two indispensable assessment activities:

- **source code verification**, which confirms that an implemented functionality works as intended; and

- **calculation verification**, which assesses the level of error that arises from various sources of error in numerical simulations as well as to identify and remove them (Thacker et al., 2004). This process mainly involves the quantification and minimization of errors introduced by the performed calculations.

Only when the executable model is verified it can be reasonably validated.

> The **validation** process evaluates the consistency of the predictive simulation outcome with the system of interest.

The validation process aims at the agreement between experimental data that defines the *ground truth* for the system of interest and the simulation outcomes. This evaluation needs to take into consideration the domain of intended application of the mathematical model as well as its expected level of agreement, since any model is an abstraction of the system of interest and only intended to match to a certain degree and for certain prescribed conditions.

2.2.3 Model Verification and Substantiation: Model Assessment in the Absence of Experimental Data

For neural network simulations, the ground truth of the system of interest can be provided by empirical measurements of activity data, for example single unit and multi-unit activity gathered by means of electrophysiological recordings. However, there are a number of reasons why this data may prove inadequate for validation. Firstly, depending on the specification of the system of interest, such data can be scarce. Secondly, even for comparatively accessible areas and assuming perfect preprocessing (e.g., spike sorting), single cell recordings represent a massive undersampling of the network activity. Thirdly, for a large range of computational neuroscientific models, the phenomenon of interest cannot be measured in a biological preparation: for example, any model relying on the plasticity of synapses within a network.

Consequently, for many neuronal network models, the most that the modeler can do with the available experimental data is to check for consistency, rather than validate in the strong sense. Thus, we are left with an incomplete assessment process. However, circumstantial evidence to increase the credibility of a model can be acquired by comparing models and their implementations against each other with respect to consistency (Thacker et al., 2004; Savio Martis, 2006). Such a technique can be meaningful in accumulating evidence of a model's plausibility and correctness even if none of the models is a *validated model* that may act as a reliable reference.

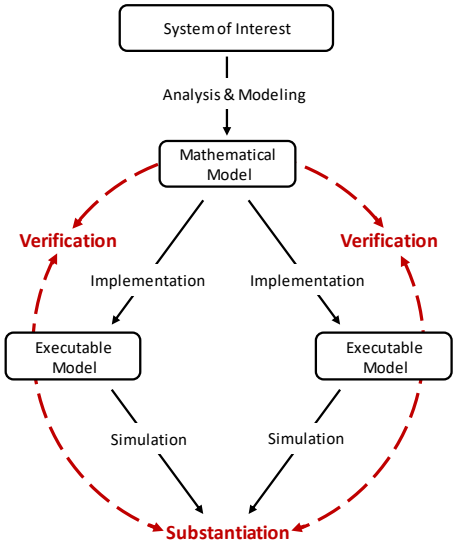


Figure 2.2 | Model verification and substantiation workflow. The workflow is derived from the model verification and validation process shown in Figure 2.1B, and can be thought of as the combination of two such processes, but without the validation against the system of interest. Instead, the consistency of the simulation outcomes of two executable models that share the same system of interest and mathematical model is evaluated in an assessment activity termed *substantiation*. Modeling and simulation activities are indicated by black solid arrows. Assessment activities are indicated by red dashed arrows.

To avoid ambiguity with the existing model verification and validation terminology, the term ***substantiation*** is proposed.

> **Substantiation** describes the process of evaluating and quantifying the level of agreement of two executable models.

Model verification and substantiation are then processes that accumulate *circumstantial evidence* of a model's correctness or accuracy by a quantitative comparison of the simulation outcomes from validated or non-validated model implementations. The interrelationship of the modeling, simulation and assessment activities are shown in Figure 2.2. To this end, the modeler has to define reasonable acceptance criteria that define the limits within which the process can be executed.

2.2.4 Application of the Terminology to Modeling and Simulation

Applying the terminology to the field of neural network modeling and simulation, the terms are used as follows.

Replication means using the author's own model, which may consist of the model source code, scripts for network generation and simulation execution as well as additional hardware and software components in a particular version (e.g., if a specific simulation software or hardware system is used). A replication should aim for bit-identicality. Although computers are deterministic, this is not always feasible; for example, if the seed of a pseudo-random number generator (PRNG) has not been recorded, or if the generated trajectory of pseudo-random numbers is dependent on the software version or underlying hardware. Beyond this, replicable models should have the property of delivering exactly the same result in successive simulations on the same hardware. When using random number generators, for example, this entails setting a seed.

A *reproduction* (or specifically, *results reproduction*) is then the re-implementation of the model in a different framework; for example, expressing a model as a stand-alone script using neural simulation tools, such as NEURON (Hines and Carnevale, 1997), Brian (Goodman and Brette, 2008), NEST (Gewaltig and Diesmann, 2007), or the SpiNNaker neuromorphic system (Furber et al., 2013), and getting statistically the same results.

2.3 Worked Example: Reproduction of a Minimal Two-Population Network Model on the Neuromorphic System SpiNNaker

In this section, the usefulness of the proposed terminology and the verification and substantiation methodology is demonstrated by a worked example. To this end, a rigorous model verification and substantiation workflow is conducted in which: (i) a published model is replicated; and (ii) reproduced on the SpiNNaker neuromorphic system.

2.3.1 Definition of the Model Verification and Substantiation Entities

According to the model verification and substantiation methodology, to execute the process, we need to define the entities: *system of interest*; *mathematical model*; and for the substantiation assessment two *executable models* (see Figure 2.2). We define as the *system of interest* the mammalian cortex. A *mathematical model* of this system has been proposed by Izhikevich (2006), who demonstrated that this model exhibits the development of polychronous groups.

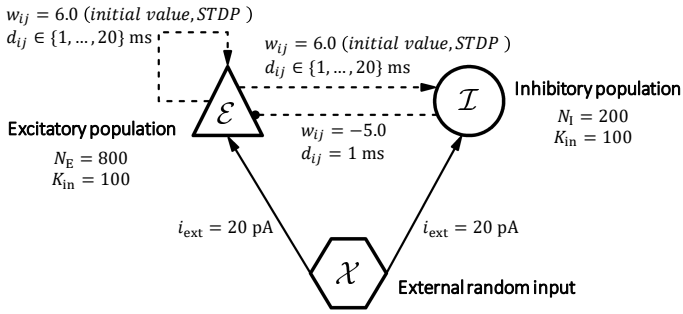


Figure 2.3 | Network topology. The minimal two-population spiking neural network described in Izhikevich (2006) consists of an excitatory and an inhibitory population of neurons, which both are stimulated by an external random input current. See the main text for description.

A C implementation of this model constitutes one of the *executable models* targeted in the verification and substantiation process, hereafter referred to as the *C model*. The second *executable model* is a reproduction of this model on the SpiNNaker neuromorphic system (Furber et al., 2013), hereafter referred to as the *SpiNNaker model*.

2.3.1.1 Mathematical Model

The model proposed by Izhikevich (2006) is a plastic minimal two-population spiking neural network consisting of an excitatory and an inhibitory population, both receiving random input from an external current source. A schematic representation of the network is shown in Figure 2.3.

Network topology

A population $\mathcal{E}$ of 800 excitatory neurons makes random connections to itself and to a population $\mathcal{I}$ of 200 inhibitory neurons with a fixed in-degree of $K_{\mathrm{in}} = 100$. The inhibitory population connects with the same in-degree, but only to the excitatory population. The excitatory connections are initially set to a synaptic strength of $w_{ij} = 6.0$ and a conduction delay drawn from a uniform integer distribution such that $d_{ij} \in \{1, 2, \ldots, 20\}$ ms. The inhibitory connections are initialized with a fixed synaptic strength and delay of $(w_{ij}, d_{ij}) = (-5.0, 1\ \mathrm{ms})$. Both populations receive input from an external source $\mathcal{X}$, which injects a current pulse of $i_{\mathrm{ext}} = 20$ pA into a randomly selected neuron every millisecond.

Component dynamics

Each neuron in the network is described by the simple neuron model published in (Izhikevich,

2003), which can reproduce a variety of experimentally observed firing statistics. The neuron model dynamics is given by the following ordinary differential equation (ODE) system:

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I(t), \tag{2.1}$$

$$\text{with } I(t) = i_{\text{syn}}(t) + i_{\text{ext}}(t), \quad i_{\text{syn}}(t) = i_{\text{exc}}(t) + i_{\text{inh}}(t)$$

$$\frac{du}{dt} = a(bv - u) \tag{2.2}$$

$$\text{if } v \geq 30\text{mV}, \text{then} \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{2.3}$$

Equations (2.1) through (2.3) describe the time evolution of a neuron's membrane potential $v(t)$ and its threshold dynamics $u(t)$. Excitatory neurons are parameterized to show a regular-spiking behavior: $(a,b,c,d) = (0.02, 0.2, -65.0, 8.0)$, and inhibitory neurons are parameterized to exhibit fast-spiking: $(a,b,c,d) = (0.1, 0.2, -65.0, 2.0)$.

The excitatory connections are plastic and evolve according to an additive spike-timing-dependent plasticity (STDP) rule:

$$w \leftarrow \begin{cases} w + A_+ \cdot \exp(-\Delta t/\tau_+) & : \Delta t \geq 0 \\ w - A_- \cdot \exp(\Delta t/\tau_-) & : \Delta t < 0 \end{cases} \tag{2.4}$$

where $\tau_+ = \tau_- = 20$ ms, $A_+ = 0.1$ mV, $A_- = 0.12$ mV, and Δt is the difference in time between the last postsynaptic and presynaptic spikes, i.e., positive on occurrence of a postsynaptic spike and negative on occurrence of a presynaptic spike. However, the rule has an unusual variant: synaptic weight changes are buffered for one biological second and then the weight matrix is updated for all plastic synapses simultaneously. Thus, synaptic weights are constant for long periods, causing the network dynamics to break down into epochs.

A comprehensive description of the model is provided in Appendix **A**.

2.3.1.2 Executable Models

C model

Various implementations of the model are available for download from the website[3] of the author of the model: a MATLAB implementation (*spnet.m*) and two versions of a C/C++ implementation (*spnet.cpp, poly_spnet.cpp*). They differ slightly in algorithms and functionality, and thus do

[3] https://www.izhikevich.org/publications/spnet.htm

not exhibit bit-identical behavior. All implementations use a grid-based hybrid time- and event-driven simulation scheme with a temporal resolution of 1 ms. Threshold detection according to Equation (2.3) is performed only at the grid points. For the numerical integration of the ODE system, a Forward Euler method is used. From the two available versions of the C/C++ implementation the computationally more precise variant *poly_spnet.cpp* was selected. It uses double precision data types and also implements an algorithm for the detection of polychronous groups. The model is implemented as a stand-alone console application.

SpiNNaker model

The SpiNNaker model was implemented in PyNN. The SpiNNaker software stack (Stokes et al., 2007) provides several neuron and synapse models as well as a model template[4] that allows users to develop custom neuron and synapse models using the event-driven programming model employed by the SpiNNaker kernel (Rowley et al., 2017). To allow the evaluation of different algorithms, the Izhikevich neuron model was implemented as a custom model using this template. SpiNNaker uses the same grid-based hybrid time- and event-driven simulation paradigm with a temporal resolution of 1 ms as the original C model implementation.

The hardware platform used for the simulation experiments was a SpiNN-3 development board. The board carries four first generation SpiNNaker chips, each containing 18 ARM968 processing cores (Temple, 2011a). For simulation control and cross-platform development, the SpiNN-3 board has to be connected to a host system, which communicates with the board using the Ethernet-based UDP[5] protocol (Temple, 2011b).

2.3.2 Definition of the Model Substantiation Assessment

In the absence of specific biological data that can define the ground truth of the *system of interest*, we are left with the simulation outcomes of the two executable models. Here, the dynamics of five selected network states in the C model are considered. Network dynamics are assessed by applying statistical methods to the network activity data, i.e., the spike trains recorded from simulations (for the statistical methods, see Section 2.3.2.2).

Note that the emergence of polychronous groups or their statistics is not used to define the ground truth, as this turns out to be rather sensitive to details not only of the mathematical model, but also of the implementational choices used to generate the executable model. For a comprehensive investigation of this aspect, see Pauli et al. (2018).

[4] The model template is available for download from the SpiNNaker repository on GitHub: `https://github.com/SpiNNakerManchester/sPyNNaker8NewModelTemplate`.

[5] The User Datagram Protocol (UDP) is a minimal message-oriented transport layer protocol.

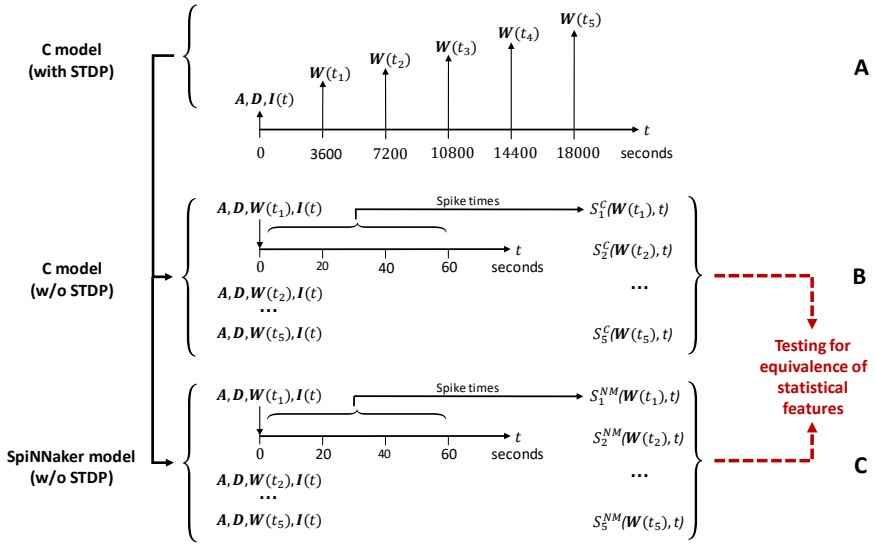


Figure 2.4 | Experimental setup. **(A)** The C model is run with STDP to define the initial conditions of five selected network states following the procedure: (i) instantiating the network and storing the connectivity matrix $\boldsymbol{A}$ and the delay matrix $\boldsymbol{D}$; (ii) selecting five points in time (here, after 1, 2, 3, 4, and 5 hours of simulated time) for which the weight matrix $\boldsymbol{W}(t_i)$ is stored, and simulating the network while recording its random input $\boldsymbol{I}(t)$. **(B)** The C model is run without STDP to generate five data sets of recorded network activity following the procedure: (i) instantiating the network with one of the sets of initial conditions $\{\boldsymbol{A},\boldsymbol{D},\boldsymbol{W}(t_i)\}$ previously created; (ii) simulating the network for 60 s simulated time and stimulating the network with the random input $\boldsymbol{I}(t)$ while recording the network activity, i.e., the spike trains of all neurons; (*iii*) repeating the procedure for all defined initial conditions. **(C)** The procedure of **(A)** is repeated, but for the SpiNNaker model. The network activity recordings $S_i^{\mathrm{NM}}(\boldsymbol{W}(t_i),t)$ of the SpiNNaker model are then compared with the network activity recordings $S_i^{\mathrm{C}}(\boldsymbol{W}(t_i),t)$ of the C model and tested for statistical equivalence of selected features, with the C model defining the ground truth.

2.3.2.1 Experimental Setup

The process of verification and substantiation is iterative and requires the repeated execution of a number of tasks. This includes preparing and conducting the substantiation assessment. Figure 2.4 illustrates the workflow, the experimental setup. Performed are a series of simulation experiments that consist of four steps.

In a first step, the initial conditions for five selected network states are defined. For this purpose, the C model is initialized with a defined seed for random number generation and is run with STDP for five hours simulated time. The connectivity matrix $\boldsymbol{A}$ and the delay matrix $\boldsymbol{D}$ of the instantiated network are stored. Five points in time are selected ($t_i:\ i=(1,2,\ldots,5)$ (here, after

1, 2, 3, 4, and 5 hours) for which the weight matrix $\boldsymbol{W}(t_i)$ is stored. During the simulation, the synaptic weights change over time according to the STDP rule, resulting in five distinct weight matrices. The random input $\boldsymbol{I}(t)$ applied to the network is also recorded. In this manner, five sets of initial network conditions are created: $\{\{\boldsymbol{A},\boldsymbol{D},\boldsymbol{W}(t_1)\},\{\boldsymbol{A},\boldsymbol{D},\boldsymbol{W}(t_2)\},\ldots,\{\boldsymbol{A},\boldsymbol{D},\boldsymbol{W}(t_5)\}\}$. Figure 2.4A illustrates this step.

Secondly, the C model is run without STDP to generate five data sets of recorded network activity. For this, the network is instantiated with a previously created set of initial conditions $\{\boldsymbol{A},\boldsymbol{D},\boldsymbol{W}(t_i)\}$. The network is then simulated for 60 s simulated time while the network activity data is recorded and the network is stimulated with the random input $\boldsymbol{I}(t)$. The procedure is repeated for all defined initial conditions. In this way, five data sets of network activity data are created for the C model: $\{S_1^{\mathrm{C}}(\boldsymbol{W}(t_1),t),S_2^{\mathrm{C}}(\boldsymbol{W}(t_2),t),\ldots,S_5^{\mathrm{C}}(\boldsymbol{W}(t_5),t)\}$. This step is illustrated in Figure 2.4B. These activity recordings define five dynamic states of the network at different stages of its evolution, constituting the reference data and fulfilling the role that ground truth data plays in a classical model validation assessment.

Thirdly, the procedure of the second step is repeated for the SpiNNaker model, resulting in a set of corresponding network activity recordings: $\{S_1^{\mathrm{NM}}(\boldsymbol{W}(t_1),t),S_2^{\mathrm{NM}}(\boldsymbol{W}(t_2),t),\ldots,S_5^{\mathrm{NM}}(\boldsymbol{W}(t_5),t)\}$. This step is illustrated in Figure 2.4B.

Finally, the network activity recordings S_i^{NM} of the SpiNNaker model are compared with the network activity recordings S_i^{C} of the C model, where they are tested for statistical equivalence of selected features.

Note that although the model parameters and properties of the two-population Inzhikevich model remain untouched, model implementations may change in successive iterations of the verification and substantiation process; consequently, so do the reference data.

2.3.2.2 Analysis of Network Spiking Activity

The degree of similarity between the different *executable models* is performed at the descriptive level of the population dynamics. Since issues such as the choice of hardware architecture, numerical precision, compiler options that affect the evaluation order of expressions, or the choice of a pseudo-random number generator and its seed should not be considered part of the mathematical model (but they are part of the executable model), it is legitimate and expected that different implementations will not result in an exact spike-for-spike correspondence (see Pauli et al. (2018) for a counterexample).

It is therefore resorted to the test for equivalence of statistical features extracted from the population dynamics. These tests were carried out in an automated, formal framework that

conducts statistical analysis of parallel spike trains using the standardized implementations found in the *E*lectrophysiology Analysis Toolkit[6] (Elephant, RRID:SCR_003833) as its backend.

When choosing the measures by which to compare the network activity, it is essential to assess diverse aspects of the dynamics. Besides widely used standard measures to characterize the statistical features of spike trains or the correlation between pairs of spike trains, this may also include additional measures that reflect more specific features of the network model (e.g., spatio-temporal patterns). Here, tests are applied that compare distributions of three statistical measures extracted from the population dynamics. They characterize the global dynamics of network activity and are calculated from the recorded spike trains. Below, these measures are outlined:

Statistical measures

Average firing rate (FR): The number of spikes n^{sp} that a neuron emits in the interval T gives its average firing rate

$$\mathrm{FR} = \frac{n^{\mathrm{sp}}(T)}{T} \tag{2.5}$$

measured in spks/s. The distribution of the firing rates in a network or population is a measure to characterize the level of network activity.

Coefficient of variation (CV) and ***local coefficient of variation*** (LV): The coefficient of variation is a standard statistical measure and defined as the ratio of the standard deviation to the mean. It characterizes the variability in a data series. The measure is applied here to the inter-spike intervals (ISIs) calculated from the ordered spike times t_i

$$\mathrm{ISI}_i = t_{i+1} - t_i. \tag{2.6}$$

The coefficient of variation is then calculated as

$$\mathrm{CV}^{\mathrm{ISI}} = \frac{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(\mathrm{ISI}_i - \overline{\mathrm{ISI}})^2}}{\overline{\mathrm{ISI}}}. \tag{2.7}$$

As an extension to the conventional *coefficient of variation*, in order to measure local variations, Shinomoto et al. (2003) introduced the *local coefficient of variation* defined as

$$\mathrm{LV} = \frac{1}{n-1}\sum_{i=1}^{n-1}\frac{3(\mathrm{ISI}_i - \mathrm{ISI}_{i+1})^2}{(\mathrm{ISI}_i + \mathrm{ISI}_{i+1})^2}. \tag{2.8}$$

[6] http://neuralensemble.org/elephant

In Equations (2.7) and (2.8) n denotes the number of neurons, and $\overline{\text{ISI}} = \frac{1}{n}\sum_{i=1}^{n} \text{ISI}_i$ defines the mean ISI. Both LV and CV analyze the relative variability in the inter-spike intervals and thus characterize the temporal structure of a spike train.

Pearson's correlation coefficient (CC): The measure quantifies the temporal correlation between all spike trains. It is determined by calculating the $n \times n$ matrix of the pairwise Pearson's correlation coefficient between all combinations of n binned spike trains.

$$\text{CC}[i,j] = \frac{< b_i - \mu_i, b_j - \mu_j >}{\sqrt{< b_i - \mu_i, b_i - \mu_i > \cdot < b_i - \mu_i, b_j - \mu_j >}} \tag{2.9}$$

In Equation (2.9), $< .., .. >$ denotes the scalar product of two vectors, where b_i and b_j are the binned spike trains, and μ_i and μ_j represent their respective means (Grün and Rotter, 2010).

The above measures are widely used in neuroscience to characterize a network's spiking activity (see, e.g., Senk et al., 2017; van Albada et al., 2018; Knight and Nowotny, 2018; Dasbach et al., 2021; Golosio et al., 2021; Heittmann et al., 2022). They can be regarded as forming a hierarchical order and evaluate different aspects of the network dynamics: rates consider the number of observed spikes, whilst ignoring their temporal structure; the coefficient of variation considers the serial correlations inherent in a spike train, whilst ignoring the relationship between spike trains; and the correlation coefficient considers coordination across neurons.

It should be noted that this conceptual hierarchy does not imply a hierarchy of failure, i.e., a correspondence on the highest level (here, the correlation of spike trains) does not automatically imply correspondence of the other measures. Therefore, it is imperative to independently evaluate each statistical property. The similarity of the distributions of these measures between simulations is evaluated using the effect size (Cohen's d), i.e., the normalized difference between the means of the distributions (Cohen, 1988). In addition to the substantiation tests selected here, more intricate comparisons can evaluate the correlation structure and dynamical features of the network activity in greater detail, as described in Gutzen et al. (2018).

2.3.3 Definition of the Model Verification and Substantiation Workflow

As defined earlier, model *substantiation* describes the process of evaluating the level of agreement between two executable models. In this respect, the method is not conclusive as to whether the model itself is correct, i.e., an appropriate description of an underlying biological reality. Consequently, the verification and substantiation workflow presented in the following sections does not evaluate any neuroscientific aspects of the model described in Izhikevich (2006).

Figure 2.5 depicts this workflow with the activities conducted; here shown in a condensed

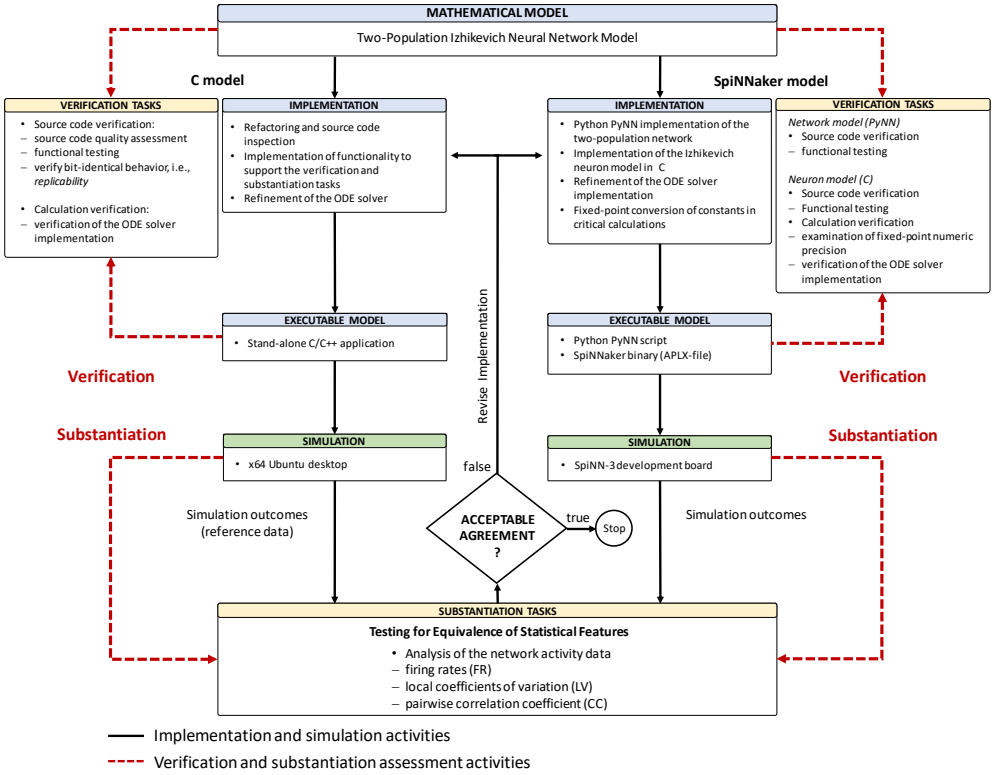


Figure 2.5 | Model verification and substantiation workflow as conducted. Condensed view of the executed workflow with the activities conducted. According to the conceptual illustration of the model verification and substantiation workflow shown in Figure 2.2, modeling and simulation activities are indicated by black solid arrows, assessment activities are indicated by red dashed arrows.

form. According to the concept of model verification and substantiation shown in Figure 2.2, it is an iterative process in which the C model and the SpiNNaker model were subjected to various implementation and verification activities. For the latter, in Section 2.2.2, two categories of activities have been defined: *source code verification*; and *calculation verification*.

Source code verification: The purpose of *source code verification* is to confirm that the functionality it implements works as intended (Thacker et al., 2004). Unlike commercially developed production software, scientific source code is used to generate results that form the basis of scientific conclusions and should, therefore, act as an available reference (Benureau and Rougier, 2017).

Calculation verification: The purpose of *calculation verification* is to assess the level of error that arise from various sources of error in numerical simulations as well as to identify and remove them. The types of errors that can be identified and removed by calculation verification are, e.g., errors caused by inadequate discretization and insufficient grid refinement as well as errors by finite precision arithmetic. Insufficient grid refinement is typically the largest contributor to error in calculation verification assessment (Thacker et al., 2004).

In the example presented here, the entire process of model verification and substantiation required three iterations of the workflow (hereafter referred to as Iteration I, II, and III) before an acceptable agreement was achieved. A complete and detailed breakdown of the activities is given in Figure 2.6, which expands on Figure 2.5 and provides a more thorough representation of the activities conducted. The substantiation activity performed at the end of each iteration is marked in Figure 2.6 with (I), (II) and (III). A summary of the substantiation assessment results is provided in Figure 2.11.

In order to be able to reproduce the findings of this work, model source codes, simulation scripts and the codes used in the verification activities are available on GitHub[7].

2.3.4 Application of the Method

In the following sections, for each of the three iterations, the verification activities are described that identified problems with the executable models, leading to consequent adaptations of the C and SpiNNaker model implementations.

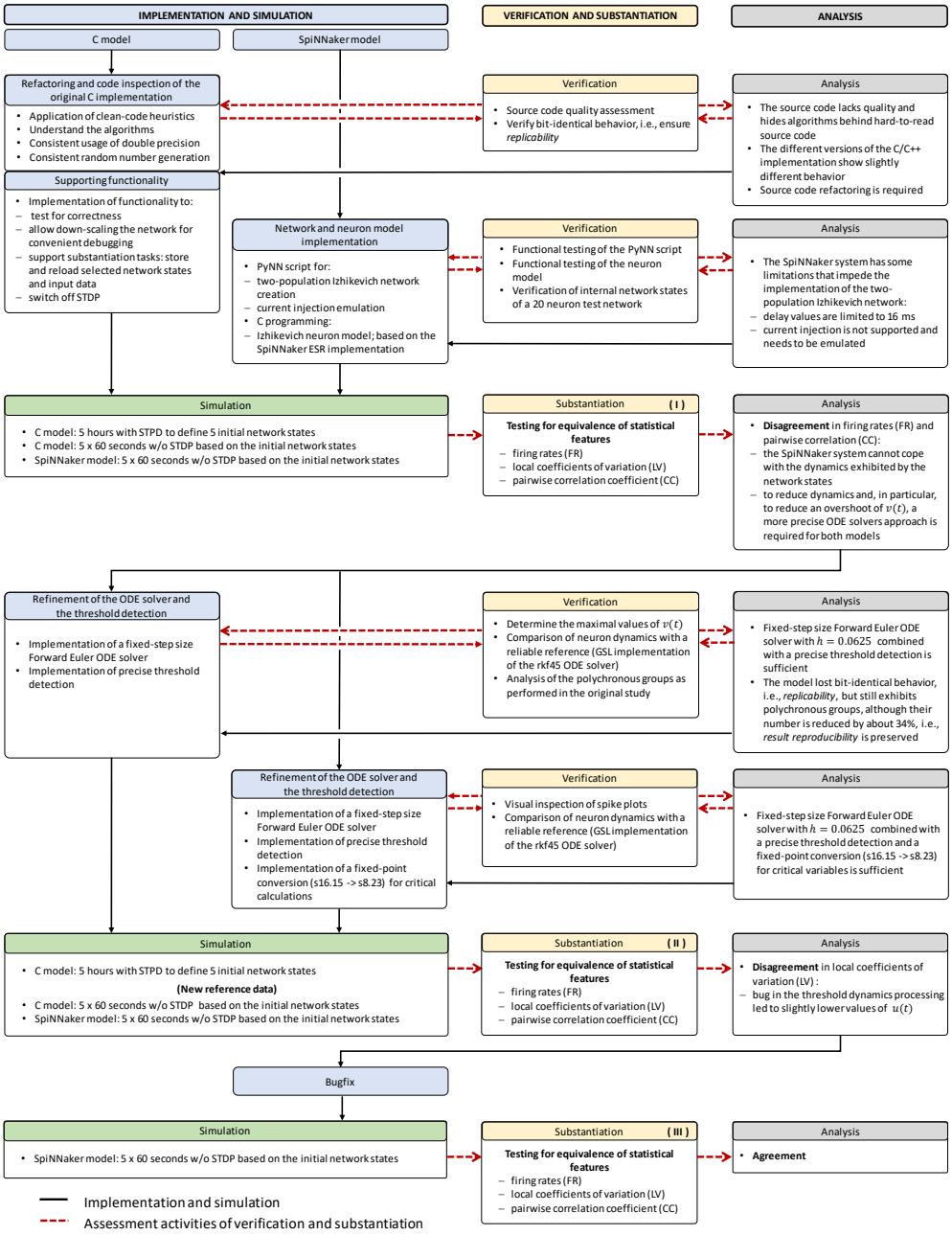
2.3.4.1 Iteration I: Source Code Verification

The primary objective of the first iteration was to verify the source codes. In the case of the C model, the emphasis was on evaluating and improving source code quality, whereas the SpiNNaker model implementation was subjected to functional testing.

C model

The original implementation (*poly_spnet.cpp*), which appears to be derived from MATLAB programming paradigms, hides the algorithms behind hard-to-read source code. To improve readability, understand the algorithms, and find potential programming and implementation errors,

[7] DOI: 10.5281/zenodo.1435831; https://github.com/gtrensch/RigorousNeuralNetworkSimulations



Caption overleaf.

Figure 2.6 | Model verification and substantiation iterations and activities as conducted. Detailed view of the activities carried out as part of the model verification and substantiation process. The flow diagram enrolls the three iterations of the workflow that were performed to achieve agreement in the substantiation assessment. The model substantiation activity performed at the end of each iteration is marked with (I), (II), and (III), corresponding to the results summary shown in Figure 2.11.

the source code was subjected to an extensive refactoring[8] and code inspection task.

The source code was fully reworked by following clean code heuristics (Martin and Coplien, 2009). Code sections related to data analysis and not part of the model itself were separated from the model implementation, but kept for functional testing. Throughout this iterative refactoring and code inspection process, care was taken to ensure that the model remained bit-identical after each iteration, ensuring replicability. The refactored source code is available on GitHub[9].

To support the experimental setup, e.g., the substantiation activities, functionality has been added to save and reload network states. STDP has also been disabled for generating the network activity data for use in the substantiation assessments (see also Section 2.3.2.1). For convenient functional testing and debugging, the implementation was also adapted to allow the two-population Inzhikevich network model to be scaled down to a 20 neuron test network. This size was chosen to be small enough for convenient manual debugging, yet large enough to exhibit spiking behavior and have a non-trivial connectivity matrix.

Performing the refactoring task not only helped to understand the C model implementation and algorithms, which is essential, but also formed the basis for the implementation of the SpiNNaker model.

SpiNNaker model

For the initial iteration of the SpiNNaker model, the Izhikevich neuron model implementation that is provided by the SpiNNaker software stack (Stokes et al., 2007) was used. This implementation employs an optimized ODE solver, which is described in Hopkins and Furber (2015), and referred to by the authors as *Explicit Solver Reduction* (ESR): *"for merging an explicit ODE solver and autonomous ODE into one algebraic formula, with benefits for both accuracy and speed."*

For network creation, simulation control and execution as well as for functional testing, Python PyNN scripts were developed that allow to conveniently execute simulations, and perform the verification and substantiation activities. Additional development work was required to circum-

[8] Refactoring – a software engineering method that belongs to the area of software maintenance – is source code transformation that reorganizes a program without changing its behavior. It improves the software structure and the readability, and so avoids the structural deterioration that naturally occurs when software is changed (Sommerville, 2015).

[9] DOI: 10.5281/zenodo.1435831; `https://github.com/gtrensch/RigorousNeuralNetworkSimulations`

vent a few restrictions of the SpiNNaker system and its software stack, namely:

The SpiNNaker framework does not allow external current injection: During each 1 ms simulation time step, an external current of $i_{\text{ext}} = 20$ pA is injected into a randomly selected neuron. This current injection is emulated by two spike source arrays forming one-to-one connections to the two populations of the network model. Those connections use static synapses, translating an external spike event into an injected current.

During the 60 seconds of simulated time, the amount of data to be stored on the SpiNN-3 board becomes too large: To limit the amount of data, a single simulation run is divided into 60 iterations. At the end of each iteration, the simulation is paused, the data is exported, and the simulation is then resumed.

For the functional testing of the PyNN scripts and the verification of the implementation of the neuron model, three approaches have been used:

Manual low level debugging on the SpiNNaker system to verify the correctness of state variables, program flow and algorithms: The SpiNNaker system provides a low level command line debugging tool called *ybug* and allows writing log information to an internal I/O buffer. The buffer is read at simulation termination and is accessible through *ybug*. This basic debugging technique was used to verify the internal states of the neuron model, the correctness of the injected current values as well as the correctness of the program flow of the implemented algorithms.

Verification of neuron dynamics using a PyNN test script that applies an external constant current to individual neurons and records the state variables: The dynamics of individual neurons resulting from an injected constant current were recorded and compared with the results obtained from a stand-alone C console application implementing the same algorithms.

Functional testing using a down-scaled version of the two-population Inzhikevich network: A down-scaled version of the network (16 excitatory and 4 inhibitory neurons) was used to verify the functional correctness of the simulation setup. Since the connectivity matrix was derived from simulations of the C model, it was also used to test the functionality added to support the activities carried out during the simulation experiments, i.e., exporting the connectivity matrix from the C model and importing it into the SpiNNaker simulation.

Substantiation assessment

For the C model and the SpiNNaker model, the simulation experiment described in Section 2.3.2.1 was conducted. The network activity recordings of the SpiNNaker model were compared with the network activity recordings of the C model, testing the equivalence of the statistical features

described in Section 2.3.2.2.

The results are summarized in the top row of Figure 2.11.The substantiation assessment reveals a significant discrepancy, most dominantly visible in the distribution of the firing rates (FR) and the pairwise correlation coefficients (CC). This mismatch, as quantified by the effect size, is consistently observed for all five reference network states (the data for all five network states is provided in Appendix **B**). Therefore, it is concluded that the models do not show an acceptable agreement and the substantiation assessment failed at the end of Iteration I.

Although the effect size is a very simple measure which only takes into account the means and standard deviations of the distributions, it provides an intuitive quantification of differences which is unbiased by the sample size. However, since the effect size cannot detect discrepancies in the distribution shape, a visual inspection is essential and additional comparison methods, such as hypothesis tests, may be needed. In Figure 2.11 it is only referred to the measures computed from 60 seconds of network activity after the fifth hour. For a visual inspection of the computed measures from the network states after 1, 2, 3, 4 and 5 hours of simulation, see Appendix **B**, Figures B1-B5.

2.3.4.2 Iteration II: Calculation Verification

The significant discrepancies in the model substantiation assessment in Iteration I suggest that there are numerical errors in one or both of the executable models. Therefore, in the second iteration, calculation verification tasks are conducted that examine the numerical integration scheme and threshold detection, as well as the 32-bit fixed-point arithmetic used on SpiNNaker.

Numeric integration scheme and threshold detection

When working with ODE systems on digital computers, it is important to make appropriate decisions regarding the choice of a numeric integration scheme. To achieve accurate approximations of their solutions, it is necessary to consider not only the form of the equations, but also the magnitude of the variables appearing in them (Dahmen and Reusken, 2005). Depending on these parameters, some ODEs may become *stiff*, requiring excessively small time steps for an *explicit* numeric integration scheme (i.e., one that uses only the values of variables at preceding time steps) to achieve acceptable accuracy and avoid numerical instability. Such equation systems require the use of an *implicit* scheme (i.e., one that finds a solution by solving an equation involving both the current values of the variables and their future values). However, implicit methods are more computationally expensive, resulting in unnecessarily long runtimes when applied to non-stiff systems (Strehmel and Weiner, 1995). The ODEs used to model neuronal behavior are often

```
EVERY MILLISECOND:
  -- neuron state update
  -- for numerical stability, two integration steps are
  -- performed for v(t)
  v := v + 0.5 · ((0.04 · v + 5.0) · v + 140.0 − u + I)
  v := v + 0.5 · ((0.04 · v + 5.0) · v + 140.0 − u + I)
  u := u + a · (b · v − u)
  -- threshold detection and spike delivery
  IF (v ≥ 30.0):
    v := c
    u := u + d
    deliverSpikeEvent()
  END
END
```

Listing 2.1 | C model: algorithm (given as pseudo-code) for updating neuronal dynamics as implemented in the original C model. The algorithm implements a fixed-step size semi-implicit symplectic Forward Euler method.

non-stiff, so an explicit numeric integration scheme is sufficient in most cases (Lambert, 1992).

The Izhikevich neuron model described by the Equations (2.1) through (2.3) is an example of such a non-stiff ODE system (see Blundell et al., 2018a). Thus, in principle, the choice of an explicit method is appropriate. Nevertheless, the integration scheme must be applied correctly, i.e., the step size must be chosen according to the desired maximum error.

The algorithm of the original C model implementation is shown in Listing 2.1. Note the symplectic, or semi-implicit Forward Euler scheme, i.e., the update of $u(t)$ is based on an already updated value $v(t)$. In an unorthodox approach, the variable $v(t)$ is integrated in two 0.5 ms steps, whereas $u(t)$ is integrated in one 1.0 ms step. The use of the Forward Euler method is appropriate here, albeit in a semi-implicit symplectic variant. The (relatively large) chosen step sizes of $h = 0.5$ ms for $v(t)$ and of $h = 1.0$ ms for the recovery variable $u(t)$, however, are questionable. More importantly, no error estimation is implemented to ensure that the integration scheme provides a reasonable approximation to the solution of the ODE system.

The spike onset of an Izhikevich neuron (all types) appears as a steep slope. The large grid-constrained threshold detection interval of 1.0 ms in the C model leads here to values of $v(t)$ well above the threshold value $\theta = 30\,\text{mV}$, where values of up to $v(t) = 1700$ mV can be observed. Figure 2.7 graphically illustrates the error introduced by this coarse approximation. This error propagates over time. According to the Equations (2.2) and (2.3), the recovery variable $u(t)$ evolves continuously, propagating the error and increasingly delaying subsequent spike events.

Moreover, the SpiNNaker system uses a non-saturating 32-bit fixed-point arithmetic. The

```
EVERY MILLISECOND:
  -- neuron state update
  REPEAT 3 TIMES:
    v := v + 0.333 · ((0.04 · v + 5.0) · v + 140.0 − u + I)
    u := u + 0.333 · a · (b · v − u)
  END
  -- threshold detection and spike delivery
  IF (v ≥ 30.0):
    v := c
    u := u + d
    deliverSpikeEvent()
  END
END
```

Listing 2.2 | SpiNNaker model: algorithm (given as pseudo-code) to demonstrate an arithmetic overflow. The algorithm implements a Forward Euler scheme similar to the implementation shown in Listing 2.1, but uses three fixed-size integration steps. The additional step increases the likelihood that $v(t)$ will take on very large values, which may lead to an arithmetic overflow of the 32-bit fixed-point data type used on the SpiNNaker system.

available value range may not be sufficient to adequately represent the values of the Izhikevich model's state variables. Depending on the algorithmic implementation, the dynamics of the model can cause fixed-point overflows, resulting in spike artifacts. The effect can be demonstrated by adding an extra integration step to the algorithm shown in Listing 2.1. This will lead to a numeric overflows of the state variable $v(t)$. Listing 2.2 shows this modification. When running this model implementation on SpiNNaker, the model produces artifacts in the form of bursts of spikes with high spike rates. Figure 2.8 shows these artifacts in a raster plot of spike events recorded from this experiment. It should be noted that the ESR implementation on SpiNNaker used in Iteration I does not produce such artifacts, but fails in adequately reproducing the network states. A more detailed examination of fixed-point numerical precision is given below.

Accuracy can be increased by adjusting the integration step size to an appropriately smaller interval – the *de facto* standard used in discrete-time digital simulations of spiking neural networks in neuroscience is 0.1 ms. However, this requires a more precise threshold detection. If threshold detection is performed only at the grid points, i.e., at intervals of 1.0 ms, the steep slope in the evolution of the membrane potential above threshold will cause $v(t)$ to quickly reach values that cannot be represented even with a double precision data type (see Figure 2.7).

To solve this problem, we can combine an exact off-grid threshold detection with a simple fixed-step size symplectic Forward Euler ODE solver that performs sub-steps. This solver variant has been implemented. The solver performs sub-steps at intervals of $h/16$ with regard

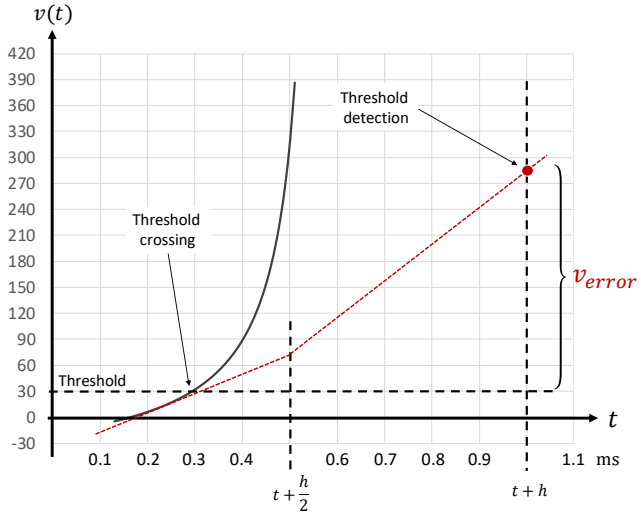


Figure 2.7 | Error caused by grid-constrained threshold detection. In the original C model, membrane potentials, i.e., values of $v(t)$, well above threshold can be observed at time of threshold detection. This introduces an error (here, indicated as v_{error}) into the Izhikevich neuron model dynamics that accumulates and propagates over time. This error then gets expressed in delayed spike times. The original C model uses a grid-constrained threshold detection interval of $h = 1.0$ ms and a semi-implicit symplectic Forward Euler method with a fixed-step size of $h/2 = 0.5$ ms. The red dashed line illustrates the calculated evolution of the membrane potential $v(t)$ when using this method. For comparison, the black solid line shows the exact evolution around threshold for a regular-spiking type Izhikevich neuron stimulated with a constant current of $i_{\text{ext}} = 5$ pA. Note the steep slope at the onset of a spike event.

to the SpiNNaker simulation resolution $h = 1.0$ ms. Spike events are forced to grid points and propagated at intervals of h. The choice of the sub-step interval was motivated by the following: firstly, $1/16$ is a power of two and can be represented in fixed-point without causing a numerical error; and secondly, it is a good compromise between the increased computational cost associated with smaller steps and the unavoidable increasing overshoot of the membrane potential resulting from larger steps. The algorithm is given as pseudo-code in Listing 2.3. The threshold detection is performed in each sub-step. Also note the multiplication with 0.0625, which avoids costly divisions. Due to the alignment of spike events to grid points, multiple spike events within a single simulation time step can potentially occur, but here are merged into a single event. However, this seems to be a very rare occurrence. Pauli et al. (2018) demonstrated that there was only a very slight change in average firing rate for this network model between a simulation locked to a 1.0 ms grid, as used here, and one carried out at a higher resolution of 0.1 ms. Therefore, this effect is considered negligible here.

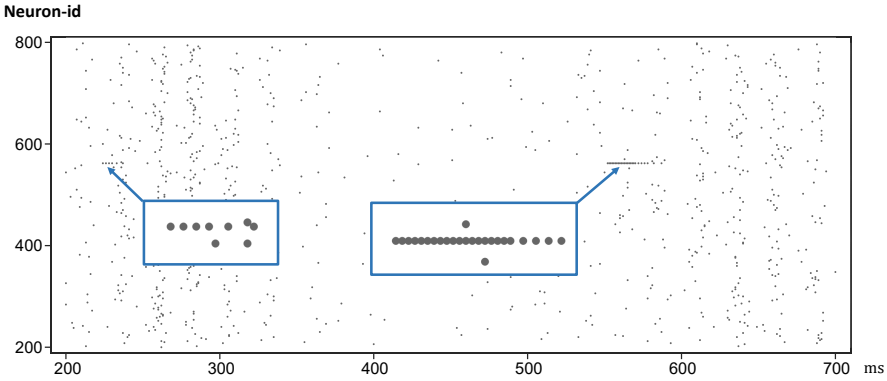


Figure 2.8 | Spike artifacts caused by fixed-point overflow. Spike raster plot obtained from a SpiNNaker simulation of the two-population Izhikevich model. Used was a symplectic Forward Euler integration scheme with a fixed integration step size of 0.333 ms and without a precise threshold detection. This integration scheme causes large values of $v(t)$ leading to an overflow of the $s15.16$ fixed-point data type. This results in simulation artifacts in the form of short spikes trains with high rates (marked by blue boxes).

In order to evaluate the accuracy of this improved ODE solver implementation and the ESR implementation provided by the SpiNNaker framework, simulations of single neurons were performed and the evolution of the membrane potentials was compared with a Runge-Kutta-Fehlberg(4, 5) (rkf45) solver implementation using the GNU Scientific Library (GSL)[10] – the working horse of differential equation solvers. The explicit rkf45 method is a good general-purpose integrator and of a higher order compared to a simple Forward Euler scheme. To serve as a reliable reference, the rkf45 algorithm was parametrized to integrate with an absolute error of 10^{-6}. The results of the single-neuron simulations are shown in Figure 2.9.

For both the fixed-step size Forward Euler and the ESR solver, spike times lag behind the rkf45 solver. Due to the accumulation of v_{error}, the lag becomes larger over time, here reaching around 20 ms capturing five spike events in 500 ms simulated time. From Figure 2.9 it can also be seen that higher spike rates lead to larger deviations; thus, the spike times of the fast-spiking type neuron are less accurate than the spike times of the regular-spiking type neuron. The error in spike timing also depends on the injected external current i_{ext}, as this also affects spike rates (data not shown). Although simpler than the ESR implementation, the fixed-step size Forward Euler scheme with sub-stepping and a precise threshold detection achieves better accuracy. However, the dynamics of the Izhikevich neuron model types are still not properly reproduced. In the

[10] https://www.gnu.org/software/gsl/

```
EVERY MILLISECOND:
  -- neuron state update
  REPEAT 16 TIMES:
    v := v + 0.0625 · ((0.04 · v + 5.0) · v + 140.0 − u + I)
    u := u + 0.0625 · a · (b · v − u)
    -- precise threshold detection
    IF (v ≥ 30.0):
      v := c
      u := u + d
      SET spikeEventHasOccurred
    END
  END
  -- spike delivery
  IF(spikeEventHasOccurred):
    deliverSpikeEvent()
  END
END
```

Listing 2.3 | SpiNNaker model: an improved algorithm (given as pseudo-code) for updating neuronal dynamics. The implementation uses a fixed-step size symplectic Forward Euler method, sub-steps, and a precise threshold detection.

following, therefore, fixed-point numeric precision is examined.

Fixed-point numeric precision

The SpiNNaker system used here is based on 32-bit ARM processors that do not provide floating-point hardware support (Furber et al., 2013)[11]. Arithmetic is implemented as fixed-point, which does not require any special hardware support. SpiNNaker stores numbers as 32-bit signed fixed-point values in the s16.15 representation. The s16.15 data type is supported by the SpiNNaker ARM C compiler toolchain.

To explain the idea of fixed-point numbers, we can state that the meaning of an n-bit binary word depends entirely on its interpretation. We can divide an n-bit word into an integer part i and a fractional part f by defining a binary point position, the *radix point*. Calculations with fixed-point numbers are performed as if the numbers are simple integers. Negative numbers are represented as two's complement. This requires an additional sign bit s; the leftmost bit, which is considered to have a negative weight. A signed fixed-point number can then be written in the positional

[11] Here used is the first generation of SpiNNaker hardware. The successor SpiNNaker2 integrates a single-precision floating-point unit (Höppner et al., 2021).

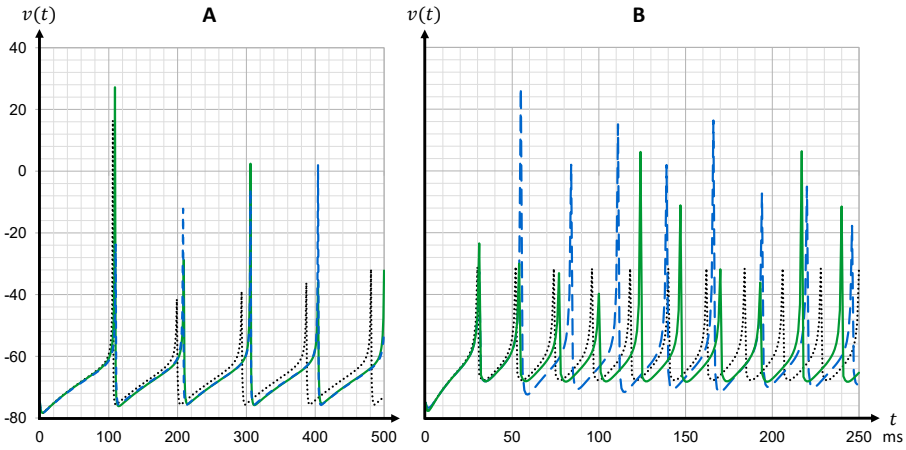


Figure 2.9 | Spike timing accuracy: comparison of different ODE solver implementations on SpiNNaker with a reliable reference. Evolution of membrane potentials $v(t)$ for different ODE solver implementations on SpiNNaker, recorded from: (**A**) a regular-spiking type; and (**B**) a fast-spiking type Izhikevich neuron. The neurons were stimulated with the constant current $i_{\text{ext}} = 5\,\text{pA}$. Neuron model dynamics were calculated using: the SpiNNaker ESR ODE solver implementation (blue dashed curves); the improved fixed-step size symplectic Forward Euler solver with sub-stepping ($h/16 = 0.0625$ ms) and precise threshold detection (green solid curves); and the GSL rkf45 ODE solver with an absolute integration error of 10^{-6} (black dotted curves), which is considered here as a reliable reference. The reference data was generated by a C program that implemented the model dynamics in double precision. For both neuron types, the ESR as well as the fixed-step size Forward Euler solver implementation on SpiNNaker show a considerable lag in the spike timing compared with the rkf45 reference solver. For the regular-spiking type neuron, both SpiNNaker implementations show almost the same error in the spike timing. A difference can be seen for the fast-spike type neuron. Here, the fixed-step size Forward Euler approach with sub-stepping and a precise threshold detection (green solid curve) shows a substantial improvement in spike timing accuracy over the ESR implementation (blue dashed curve).

number representation as

$$B = sb_{i-1}b_{i-2}\ldots b_1 b_0.b_{-1}b_{-2}\ldots b_{-f}, \tag{2.10}$$

where the value of B is given by

$$x(B) = -s2^i + \sum_{k=-f}^{i-1} b_k 2^k. \tag{2.11}$$

The value range of a fixed-point data type is small compared to a single- or double-precision float type of the same bit size. For signed fixed-point types, i.e., types of the form $si.f$, this value

range is given as

$$-2^i \leq x(B) \leq 2^i - 2^{-f}. \tag{2.12}$$

Accordingly, the 32-bit s16.15 data format used on SpiNNaker, which uses $i = 16$ bits for the integer part, $f = 15$ fractional bits, and a sign bit, provides the following value range

$$-2^{16} = -65536 \quad \text{to} \quad 2^{16} - 2^{-15} = 65535.999969482. \tag{2.13}$$

This is a very limited range, increasing the likelihood of fixed-point underflow and overflow conditions. Moreover, arithmetic does not saturate on SpiNNaker (Hopkins and Furber, 2015). In case of an arithmetic overflow, the value *wraps around*; i.e., adding two positive numbers will then result in a negative number. In simulations, this may be seen as spike artifacts (see Figure 2.8).

Another consequence of the 32-bit fixed-point arithmetic, and an additional source of numerical inaccuracy, is that not every number can be represented with sufficient precision. For example, although small, the error in the s16.15 representation of the constant value 0.04 in Equation (2.1) induces a noticeable delay in the spike timing. This can be demonstrated with a few calculations.

To represent a number in $si.f$, its value is shifted f bits to the left, i.e., multiplied by 2^f. The conversion of the value 0.04 into the s16.15 representation gives

$$0.04 \cdot 2^{15} = 1310.72_{(\text{s16.15})}. \tag{2.14}$$

The SpiNNaker ARM C compiler truncates the fraction and stores the number as a 32-bit value, which results in (here given as a hexadecimal number)

$$0\text{x}0000051\text{E}. \tag{2.15}$$

This value is used in calculations and is slightly smaller than the original number, as a back conversion reveals

$$1310_{(\text{s16.15})} \cdot 2^{-15} = 0.03997802. \tag{2.16}$$

In order to demonstrate the effect on the dynamics of the Izhikevich model, for an arbitrary chosen set of initial values, we can calculate dv/dt using Equation (2.1). We can compare the result obtained from a calculation using the precise constant value 0.04 with the result obtained from a calculation using the slightly imprecise s16.15-converted value 0.03997802. As set of values, we choose $v = -75$ mV, $u = 0$ mV, and $I = 0$ pA. The calculation using the exact constant value

results in

$$0.04 \cdot 75^2 + 5 \cdot (-75) + 140 = -10.0000000. \tag{2.17}$$

Using the s16.15-converted value, the calculation yields

$$0.03997802 \cdot 75^2 + 5 \cdot (-75) + 140 = -10.1236357. \tag{2.18}$$

This slightly more negative values cause threshold crossing to occur later, and thus delay the times spike events occur. The effect can be mitigated if such critical calculations are performed using a fixed-point data type that provide higher precision employing a larger number of factional bits; for example, a data type with $f = 23$. This reduces the distance between two consecutive fixed-point numbers; the unit in the last place (ulp). Converting the value 0.04, for example, into an s8.23 representation gives

$$0.04 \cdot 2^{23} = 335544.32_{(\text{s8.23})}, \tag{2.19}$$

and converting the value back results in

$$335544_{(\text{s8.23})} \cdot 2^{-23} = 0.039999962. \tag{2.20}$$

Repeating the calculation of dv/dt shows a significant improvement in accuracy

$$0.039999962 \cdot 75^2 + 5 \cdot (-75) + 140 = -10.00021375. \tag{2.21}$$

This approach has two disadvantages. Firstly, the reduced value range of the s8.23 data type, which is

$$-2^8 = -256 \quad \text{to} \quad 2^8 - 2^{-23} = 255.999999881. \tag{2.22}$$

Secondly, the s8.23 data type is not available on SpiNNaker, i.e., it is not supported by the ARM C compiler toolchain. However, these limitations can be worked around. It will be demonstrated below that the data type can be mimicked through a few calculations involving simple type conversions. In addition, by using the proposed fixed-step size symplectic Forward Euler method with sub-steps and precise threshold detection, as well as using a specific ordering of the arithmetic operations, it can also be ensured that values stay within the value range of the s8.23 data type.

The ARM C compiler converts a constant placed in the C code into the s16.15 representation. Therefore, we must let the value $335544.32_{(\text{s8.23})}$ (see Equation (2.19)) appear as a s16.15 constant.

We can write

$$335544.32_{(s16.15)} = 10.24 \cdot 2^{15}. \tag{2.23}$$

To get the original value, a right-shift operation of 8 bits is required

$$10.24 \cdot 2^{-8} = 0.04. \tag{2.24}$$

Note that $2^{-8} = 0.00390625$ has an exact representation in s16.15. Also note that multiplying 10.24 with the power of two of the membrane potential may cause an arithmetic overflow of the s16.15 data type. Therefore, the order in which the operations are performed must also be taken into account. If we apply the proposed type conversion to Equation (2.1), we can rewrite the equation as

$$\frac{dv}{dt} = ((10.24 \cdot v) \cdot 0.00390625)) \cdot v + 5 \cdot v + 140 - u + I(t). \tag{2.25}$$

The order of operations here ensures that no numeric overflow occurs. The corresponding algorithm is shown in Listing 2.4.

The above also applies to the Izhikevich neuron model parameters a and b, which add an error to the recovery variable $u(t)$. Further, the example ignored that the neuron model state variables $v(t)$ and $u(t)$ are themselves fixed-point numbers that add a numerical error.

In the course of the implementation of the Izhikevich neuron model on SpiNNaker, and the adaptations of the model during the verification and substantiation process, fixed-point data type conversion was added to all constant values involved in critical calculations; the constant value 0.04 in Equation (2.1) and the neuron model parameters a and b in Equation (2.2). To investigate the effect, regular-spiking and fast-spiking type Izhikevich neurons with and without fixed-point data type conversion were simulated. The evolution of the membrane potentials was compared with equivalent model implementations using the Runge-Kutta-Fehlberg(4, 5) solver from the GNU Scientific Library (GSL); analogous to the comparison of the different integration schemes (Figure 2.9). The results are shown in Figure 2.10. For both neuron model types, a substantial improvement in the spike timing is achieved. The dynamics of the regular-spiking type neuron is very close to the rkf45-reference when using the solver implementation that employs data type conversion. Here, the spike timing accuracy of the regular-spiking type neuron still lags slightly behind the rkf45 reference. This can be explained by the still existing overshoot in $v(t)$ at threshold detection (even if it is small), and the higher firing rate of the neuron type. The delay in spike times propagates over time, and hence increases with the spike rate.

```
EVERY MILLISECOND:
  -- neuron state update
  REPEAT 16 TIMES:
    A := 10.24 · v
    A := A · 0.00390625
    A := A · v
    B := 5.0 · v + 140.0 − u + I

    v := v + 0.0625 · (A + B)
    u := u + 0.0625 · a · (b · v − u)

    -- threshold detection
    IF (v ≥ 30.0):
      v := c
      u := u + d
      SET spikeEventHasOccurred
    END
  END

  -- spike delivery
  IF(spikeEventHasOccurred):
    deliverSpikeEvent()
  END
END
```

Listing 2.4 | SpiNNaker model: the same algorithm (given as pseudo-code) as shown in Listing 2.3, but adds fixed-point conversion to the constant 0.04. In order to prevent the compiler from optimizing the code and perhaps arranging the operations in an inappropriate order, the critical calculations in the Equation (2.25) are placed in separate lines. Note that suppressing optimization in this way works for the ARM C compiler (it has been verified through an analysis of the generated assembler source code), but cannot be generalized.

Substantiation assessment

As the C model was adapted during Iteration II, we can no longer speak of a *replication*. Therefore, before performing the model substantiation assessment, it needs to be verified whether the results of the modified model are compatible with the original, i.e., whether or not *result reproducibility* is preserved. For this purpose, the development of polychronous groups in the modified C model was evaluated using the analysis provided in Izhikevich (2006). This evaluation showed that the number of polychronous groups is reduced by about 34%. Thus, it still exhibits the behavior reported in the original manuscript (Izhikevich, 2006), albeit in a weakened form. Since Pauli et al. (2018) demonstrated that the number of polychronous groups developed by the C model varies significantly with implementation details, the improvement in numeric precision made here aligns with this observation. As such, this result is expected and considered to fall within the

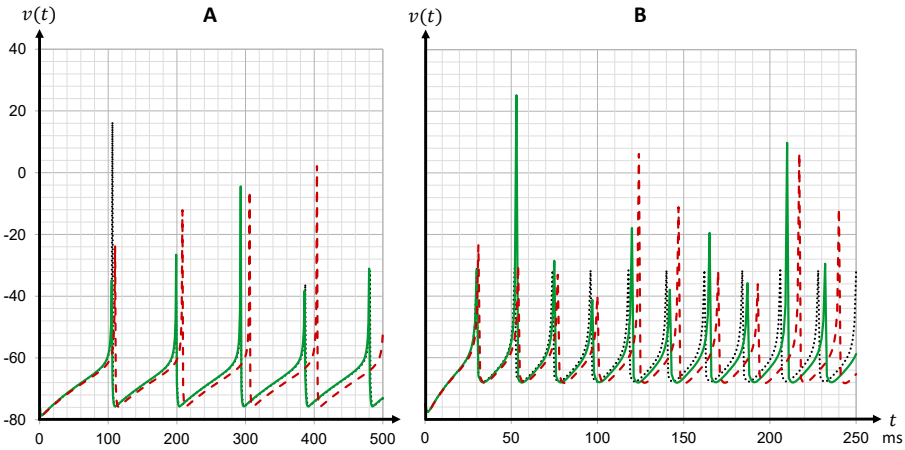


Figure 2.10 | Spike timing accuracy: comparison of two implementations of an ODE solver, with and without fixed-point data type conversion. Evolution of membrane potentials $v(t)$ for different numerically precise versions of an ODE solver, recorded from: (**A**) a regular-spiking type; and (**B**) a fast-spiking type Izhikevich neuron. The neurons were stimulated with a constant current of $i_{\text{ext}} = 5\,\text{pA}$. Neuron model dynamics were calculated using the fixed-step size symplectic Forward Euler implementation with sub-stepping ($h/16 = 0.0625$ ms) and a precise threshold detection. The solver was implemented in two different versions: using the s16.15 fixed-point data type (red dashed curves); and with converting the s16.15 data type into an s8.23 representation (green solid curves). For comparison, the model dynamics were also calculated using the GSL rkf45 ODE solver with an absolute integration error of 10^{-6}, which is considered here as a reliable reference (black dotted curves). For both neuron types, a substantial improvement in the spike timing accuracy can be observed. For the regular-spiking type neuron, it is even nearly close to the rkf45 reference.

acceptance criteria.

The change in the C model implementation requires the initial states to be regenerated in order to perform the substantiation assessment as described in Section 2.3.2. This procedure was repeated for this iteration. The result of the assessment is shown in the middle row of Figure 2.11.

The improved ODE solver implementation, used in both models, leads to a good match in the firing rates (FR) and the pairwise correlation coefficients (CC). It is noted, though, that the distributions are shifted from those expressed by the C model implementation in Iteration I. The shift of cross-correlation to lower values may well account for the smaller number of polychronous groups developed. Both the firing rates and the cross correlations also show small effect sizes after this iteration. In case of the CC distributions, the effect size has to be interpreted with care, as it assumes Gaussian-like distributions which is clearly violated by the bimodality of the CC distributions. Nevertheless, in combination with visual inspection and additional comparison

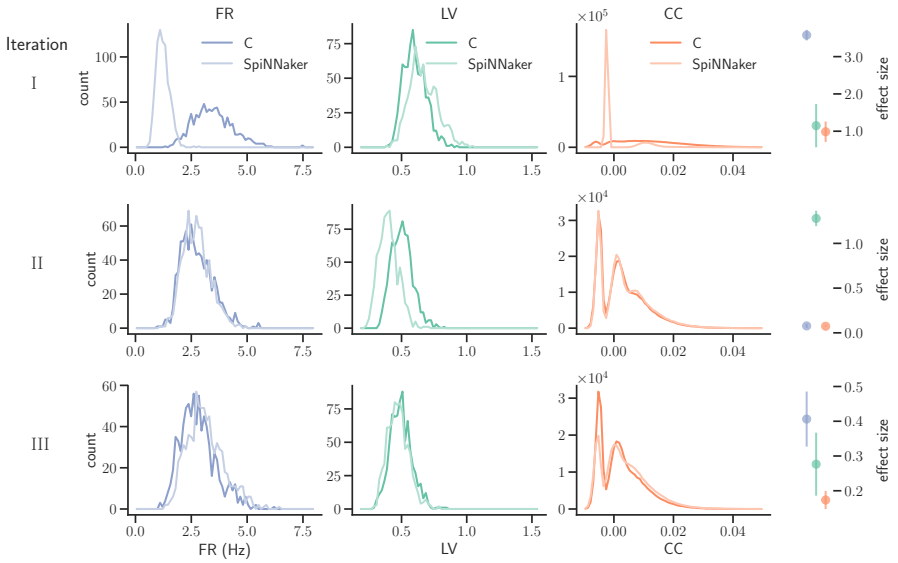


Figure 2.11 | Model substantiation assessment based on spike data analysis. Histograms (70 bins each) of the three characteristic measures computed from 60 s of network activity after the fifth hour of simulation: left, firing rates (FR); middle, local coefficients of variation (LV); right, pairwise correlation coefficients (CC). For FR and LV, each neuron enters the histogram, for CC each neuron pair. Results are shown for three iterations (rows) of the substantiation process of the C model (dark colors) and SpiNNaker model (light colors), see Figure 2.6. On the far right, the difference between the respective distributions is quantified by the effect size: the graph shows the mean and standard deviation effect size calculated for each of the five network states (after 1, 2, 3, 4, and 5 hours of simulation; see also Appendix **B**).

measures, its application here provides a useful discrepancy quantification.

A discrepancy can still be seen between the distributions of the coefficients of variation (LV). The distribution for the SpiNNaker model is shifted toward lower values, indicating a higher degree of regularity than that of the C model. This is confirmed by the consistently high effect size obtained for the five reference network states. Therefore, it is concluded that there is still a disagreement in the executable models, and that model substantiation has not been achieved at the end of Iteration II.

2.3.4.3 Iteration III: Resolving an Implementation Issue

The slight discrepancy in regularity observed in Iteration II allowed to identify systematic differences in spike timing between the two models, hinting at an error in the numerical integration

of the single neuron dynamics. Indeed, the visual comparison of the dynamics of individual neurons on SpiNNaker with a stand-alone C application that implements an identical fixed-step size symplectic Forward Euler ODE solver, revealed a small discrepancy in the sub-threshold dynamics, leading to a fixed systematic delay in the spike timing. An implementation issue in the precise threshold detection algorithm was identified as the cause.

Substantiation assessment

The achieved result after resolving the issue and repeating the SpiNNaker simulations is shown in the bottom row of Figure 2.11. A close match of all three distributions can be observed, consistently across the five reference network states (the data is provided in Appendix **B**, bottom row in Figures B1-B5). The comparison is not perfect, with the distribution of firing rates showing the largest discrepancy with only a subtle shift toward higher firing rates for the SpiNNaker simulation. The small discrepancies between the two implementations are quantified by the effect size, and demonstrate that a considerable reduction of the mismatch is achieved. All effect sizes are classified in the range of small to medium according to Cohen (1988). While further iterations of the model implementation in the verification and substantiation process may further improve the effect size scores, for the purposes, here the remaining mismatch is judged to be in the range of acceptable agreement. Therefore, it is concluded that the executable models are in close agreement at the end of Iteration III.

2.4 Discussion

Terminology

Clear terminology is essential for effective communication. Well defined terminology avoids ambiguity and is a factor of quality. The proposed terminology adapts the existing ACM (Association for Computing Machinery, 2016) terminology for reproducibility and replicability, as it seems most appropriate for the purposes. Alternative definitions exist, and terminology for research reproducibility is an ongoing theme of a controversial debate. The application of methodologies from model verification and validation to the field of neural network modeling and simulation can be of great value, but some adaptations were suggested that may fit the domain better. In particular, the terms *mathematical model* and *executable model* proposed instead of the terms *conceptual model* and *computerized model*, are intended to yield better separation of the entities they describe, so that, for example, implementation details are not falsely understood to belong to the mathematical model. This is important, as the classic *'one model – one code'* relationship does not typically apply to spiking neuron network models. Instead, they

are implemented using general-purpose neural simulation tools such as NEURON (Hines and Carnevale, 1997), Brian (Goodman and Brette, 2008) or NEST (Gewaltig and Diesmann, 2007), which can run many different models. This can also be extended to neuromorphic hardware. In addition, model simulation codes may be partially generated by other tools (Blundell et al., 2018b). This scenario abstracts the implementation details away from the modeler, who can focus on analysis and modeling, and has the further advantage that individual components, such as neuron and synapse models, can be verified separately and can later serve as reliable references. Further, the proposed terminology, which aims to better express the underlying intent in the domain of neuroscience modeling and simulation, may also help pave the way for a more formalized approach to model verification and validation in this field.

Model verification and substantiation

The introduced concept of *model verification and substantiation* can be found used in a number of studies, but without strict definition (see e.g., van Albada et al., 2018; Knight and Nowotny, 2018; Golosio et al., 2021). Here, a definition has been introduced in accordance with the study of Gutzen et al. (2018). We found the term *model substantiation* to be an appropriate choice, because it expresses the intent of creating circumstantial evidence of a model's correctness with respect to a reliable reference, i.e., a ground truth to be defined. Consequently, a model substantiation assessment is not conclusive as to whether a model represents an appropriate description of an underlying biological reality (the *system of interest*).

The proposed methodology has a number of advantages. Firstly, from the point of view of computational neuroscience, simulation results should be hardware independent, at least at the level of statistical equivalence. In practice, implementations can be sensitive to compilers, software versions, and also hardware architecture. The physical hardware used to simulate a model should therefore also be considered part of the model implementation. Applying the proposed model substantiation methodology allows a researcher to discover and correct implementation weaknesses. Secondly, in the case of new types of hardware, which includes novel neuromorphic architectures, the methodology can help build confidence, but also uncover shortcomings as demonstrated by the worked example. Finally, in neuroscience, models often function as discovery tools and hypothesis generators in cases where experimental data, against which a model could be validated, does not exist. Performing a substantiation assessment is an option to accumulate circumstantial evidence for a model's plausibility and self-consistency, although it cannot reveal whether a model reflects reality.

Worked example

The application of the terminology and methods has been demonstrated by means of a worked example and the execution of a rigorous workflow. Assessed was the level of agreement between the C implementation of the spiking network model proposed by Izhikevich (2006) and a reproduction of its underlying mathematical model on the SpiNNaker neuromorphic system. The use of this network was motivated by its unorthodox implementation choices, examined in greater detail in Pauli et al. (2018). These issues make it a particularly illustrative example for a reproduction on the SpiNNaker neuromorphic system and to demonstrate various aspects of source code and calculation verification.

In the process of executing the workflow a number of standard methods from software engineering were used. This discipline is concerned with the *"application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software"* (Bourque and Fairley, 2014). Such methods include, for example, the application of clean code heuristics, test driven development, continuous integration and agile development methodologies, with the common goal of building quality into software. The conducted workflow should also be seen in this context. Despite the increasing recognition of Research Software Engineering (RSE) as a key discipline, it is noted here that software engineering methods, although critical for the development of high quality software, are still underutilized in computational science in general and in computational neuroscience in particular. For the investigated network model, it is important to emphasize that the awareness of software engineering methodologies was even less widespread at the time of publication, and so the yardsticks for source code quality applicable by today's standards should be considered in their temporal distance. Credit must in any case be given for the unusual step of publishing the source code, allowing scientific transparency and making studies such as the current one, that of Gutzen et al. (2018), and Pauli et al. (2018), possible.

Level of agreement

At the end of the third iteration of the executed workflow, based on the substantiation assessment, it was concluded that the executable models are in acceptable agreement. This conclusion is predicated on the domain of application and the expected level of agreement that was defined for three characteristic measures of the network activity. It should be emphasized here that these definitions are set by the researcher: further iterations would be necessary, if, for example, a level of agreement is set requiring a spike-for-spike reproduction of the network activity data, as applied by Pauli et al. (2018).

We speculate here that the remaining discrepancy in the statistical measures at the end of

Iteration III can be explained by the loss in precision when converting the C model's double precision weight matrix into the fixed-point representation used by the SpiNNaker system. Due to truncation, the absolute values of synaptic weights after conversion are always slightly smaller than their higher precision origin; hence increasing the values of negative weights, contributing to a minimal shift toward higher firing rates (see Figure 2.11).

Both the original C model implementation and the SpiNNaker system use a spike timing resolution of 1 ms. Spikes are propagated at this interval while neuron dynamics are advanced by performing sub-steps. Here, the substantiation assessment cannot give us any further insight into whether the temporal resolution of spike propagation is sufficient – both models are affected. In Section 4.5, this question is re-visited and it is shown that it is sufficient for this model to propagate spikes at this interval.

Transferability of the method

Although some of the applied verification tasks are closely tied to model implementation details (e.g., functional testing), the presented methodology is transferable to similar modeling tasks, where workflows can also be further automated. The quantitative comparison of the statistical measures carried out in the substantiation assessment was performed using the modular framework `NetworkUnit`[12] (NetworkUnit, RRID:SCR_016543), an open source Python module, presented in (Gutzen et al., 2018). `NetworkUnit` facilitates the formalized application of standardized statistical test metrics that enable the quantitative validation of network models on the level of the population dynamics. It is stressed here the importance of using a common tool to extract the statistical features for both simulation outcomes in the substantiation procedure in order to prevent distortions in the substantiation results due to discrepancies in the implementations of the substantiation procedure itself. In addition, making use of methods provided by such open-source projects greatly contributes to the correctness and replicability of the results.

Adherence to formalized processes fosters transparency and comprehensibility and reduces the risk of incorrect conclusions. Moreover, simulation tools as well as neuromorphic hardware platforms can benefit from formalized and automated verification and validation procedures, so that their reliability can be inherited by user-developed models that are simulated using these tools and frameworks. Most importantly, such standardized procedures are designed not to place an additional burden on researchers, but rather to open up simple avenues for computational neuroscientists to increase the rigor and reproducibility of their models. Simulation and analysis tools, frameworks and collaboration platforms are part of the research infrastructure on which scientists base their work, and thus should meet high software development standards. Software

[12] https://github.com/INM-6/NetworkUnit

engineering methods, including the model verification and substantiation workflow presented in the previous sections, as well as verification and validation methods in general, need to become a mainstream aspect of computational neuroscience.

Chapter 3

A System-on-Chip Based Hybrid Neuromorphic Compute (HNC) Node Architecture for Reproducible Hyper-Real-Time Simulations of Spiking Neural Networks

"The enjoyment of the tools one works with is, of course, an essential ingredient of successful work."

Donald Knuth, The Art of Computer Programming, Volume 2

3.1 Introduction

In recent years, the technology of programmable logic devices and the associated tools for their application have advanced greatly, benefiting from the continuous developments in semiconductor technology. In particular, attractive new opportunities for architecture designs have emerged from developments in field-programmable gate arrays (FPGAs) empowered with general-purpose processors. These devices integrate a programmable logic device together with a complete computer system into a single chip, a System-on-Chip (SoC).

Presented in this chapter is a novel FPGA-SoC based hybrid hardware and software mixed architecture for a neuromorphic compute node (henceforth *HNC node*) intended to operate in a multi-node cluster configuration. The HNC node design builds on the AMD Xilinx Zynq-7000 SoC device architecture (AMD Xilinx, 2018), which integrates an ARM-based general-purpose processor and a powerful FPGA. The proposed architecture leverages both components and takes advantage of their tight coupling.

The development pursued several objectives, with the primary goal being the design and prototypical implementation of a neuromorphic compute node that can form the foundation for a flexible platform that enables accelerated and reproducible simulations of spiking neural networks in neuroscience. This primary goal defined the boundary conditions for a strictly neuroscience requirement-driven design. The development is thus to be seen as a complementary yet distinct approach to the neuromorphic developments aiming at brain-inspired and highly efficient novel computer architectures for solving real-world tasks.

Design space exploration, the elaboration and evaluation of architecture variants, and trade-off analysis were central tasks during development, with performance as the key objective. To achieve best possible performance and push the technology to its limits, component designs have been optimized for low latency. In this regard, the development has also been an exploration of commercial off-the-shelf FPGA-SoC device technology and tools.

This chapter is organized as follows. Firstly, the development environment is described. A brief introduction to the development platform and the Zynq-7000 SoC device architecture is given, and the development approach and digital design methodology is explained. After this introductionary part, secondly, the HNC node architecture is presented. This begins with an introduction to the conceptual ideas and provides an overview of the system architecture. This is followed by the chapter's two main sections, which present the software system of the HNC node and the microarchitecture of hardware blocks. Technical principles and design choices are explained, and architecture alternatives are discussed and evaluated. Finally, the HNC node's timing behavior and operation latencies are described, which build the basis for a systematic

performance assessment presented in Chapter 5. An overview of chip resource utilization and power consumption is also provided.

Contributions

- A novel FPGA-SoC-based hybrid hardware and software mixed neuromorphic compute node architecture is presented. The development was strictly driven by neuroscience requirements and is to be seen as a complementary yet distinct approach to the neuromorphic developments aiming at brain-inspired and highly efficient novel computer architectures for solving real-world tasks.
- The development demonstrates the suitability of commercial off-the-shelf FPGA-SoC technology as a substrate for neuromorphic computing for application in computational neuroscience. The technology provides a good compromise between flexibility and efficiency, and enables novel architecture designs that can meet the demanding requirements of neuroscience modeling and simulation.
- The development revealed the technical challenges posed by the field of application and the aspects that demand special attention, both from an architectural and technological point of view. Regarding the latter, development and evaluation was also an exploration of commercial off-the-shelf FPGA-SoC technology. It has been found that memory architecture, which is closely linked to semiconductor technology, plays a key role, imposing design constraints that largely determine performance and possible system size, i.e., it affects the scalability of a system.

3.2 Development Environment

3.2.1 Development Platform

The technical implementation of the HNC node prototype was carried out using the AMD Xilinx Zynq-7000 SoC ZC706 Evaluation Kit. The development board carries an XCZ7045 chip from the AMD Xilinx Zynq-7000 SoC device family (AMD Xilinx, 2018). The XCZ7045 integrates a dual-core ARM Cortex-A9 processor and a freely programmable and re-configurable logic device – a field programmable gate array (FPGA). Both the processor and the FPGA can access a 1GiB DDR3 external memory. The development board also provides a standard set of peripherals (e.g.,

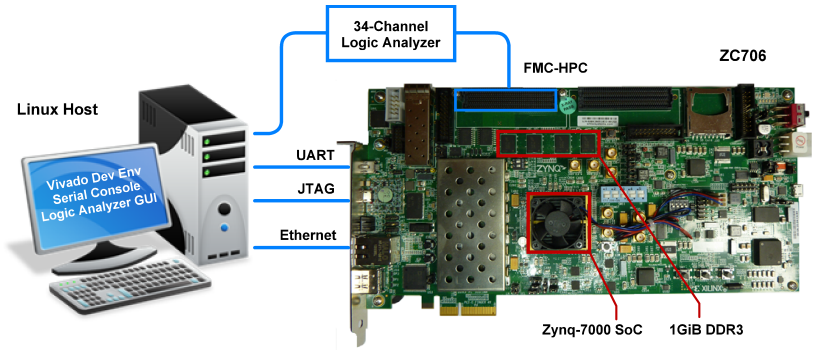


Figure 3.1 | Setup of the development and test environment. The ZC706 development board is connected to a Linux host system. The host system provides the AMD Xilinx Vivado development environment, a serial console, and the logic analyzer GUI front end.

JTAG[1], UART[2], PCIe[3], Ethernet, etc.) and expansion connectors (e.g., FMC-HPC[4] and FMC-LPC, etc.). Some of these peripherals are used in the development and test environment setup, where they connect the board to a Linux host system. The setup is shown in Figure 3.1. The Linux host system provides the software frameworks and tools for development, test, and operation of the HNC node. For the development process, i.e., hardware-software co-development, design synthesis, and analysis, the AMD Xilinx Vivado 2019 Design Suite (AMD Xilinx, 2019e) and the AMD Xilinx Vivado 2019 Software Development Kit (SDK) and embedded system tools (AMD Xilinx, 2019b) were used. These tools use the JTAG interface of the board to program and debug the XCZ7045 chip. The implemented design of the HNC node itself requires an Ethernet connection and a serial UART console interface (TTY[5] over serial on USB) for operation and user interaction. For debugging purposes and timing analysis, the development board was also connected to an external 34-channel logic analyzer via an FMC-HPC expansion port.

3.2.2 AMD Xilinx Zynq-7000 SoC Device Architecture

The AMD Xilinx Zynq-7000 SoC devices (AMD Xilinx, 2021b) are feature-rich and very complex. Therefore, in the following, I will only give a brief overview of the fundamental device architecture and its building blocks, emphasizing the essential aspects needed for further

[1] JTAG is an industry standard named after the Joint Test Action Group.
[2] Universal Asynchronous Receiver Transmitter
[3] Peripheral Component Interconnect Express (PCIe) is a high-speed serial computer expansion bus standard.
[4] FPGA Mezzanine Card (FMC) is an ANSI standard that defines I/O modules. It allows for two sizes of connectors, high pin count (HPC) and low pin count (LPC).
[5] A TeleTYpewriter, or TTY, is a text-only console.

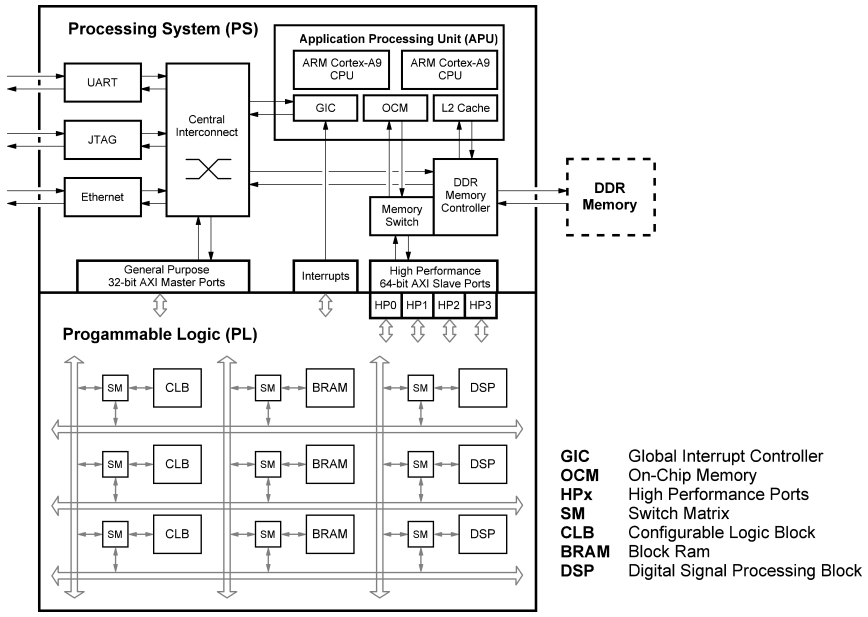


Figure 3.2 | Basic architecture of an AMD Xilinx Zynq-7000 SoC device. The Zynq-7000 series devices comprise the two sections: Processing System (PS), and Programmable Logic (PL). The device architecture is complex. Shown here is a reduced view with the main building blocks. The PS integrates most components of a computer. Here, the central component is the Application Processing Unit (APU) featuring a dual-core ARM processor. The PL provides a powerful FPGA. See main text for description.

understanding.

Figure 3.2 shows the basic architecture of such a device. It consists of two main sections: a Processing System (PS); and a Programmable Logic (PL) part. The PS features a dual-core ARM Cortex-A9 processor (up to 1GHz) as part of an Application Processing Unit (APU), a fully integrated memory controller, and multiple peripherals; a selection is shown in Figure 3.2, here JTAG, UART, and Ethernet. For coupling with the PL, several PS-PL interfaces are provided: e.g., four 64-bit high-performance ports (HP0 to HP3); and two 32-bit general-purpose ports. These interfaces follow the AXI[6] standard and enable efficient data exchange between PS and PL. The PS also provides a number of PS-PL and PL-PS interrupt ports, which can be used to synchronize the operation between the PS, the PL, and the peripherals. A central interconnect fabric allows all components to communicate and exchange data.

In addition to integrating most of the components of a computer into a single chip, Zynq-

[6] Advanced eXtensible Interface (AXI) is an open bus architecture standard.

7000 SoC devices also include an FPGA (the PL section of the chip), which can be freely programmed with custom logic and re-configured. The basic structure of an FPGA is a matrix of configurable logic blocks and memories that can be arbitrarily interconnected (illustrated in the lower half of Figure 3.2). The XC7045 chip here used for the implementation of the HNC node provides 350,000 configurable logic blocks (CLBs), 218,600 look-up tables (LUTs), 437,200 flip-flops (FFs), 19.2 Mbit of fast static block RAM (BRAM), which can be customized for different configurations, and 900 digital signal processing (DSP) blocks for the implementation of arithmetic operations. The resources provided are equivalent to those of a Kintex-7 FPGA (AMD Xilinx, 2018).

Through the close integration of components and efficient communication interfaces, the Zynq-7000 SoC SoC architecture achieves a tight coupling between the PL and the PS. Using the high performance ports, custom logic on the PL can also direct access the external memory (shown on the right in Figure 3.2) without APU intervention, sharing data with the APU in the processors address space. The ability of the architecture to offload compute-intensive tasks from a CPU to programmable logic allows for designs that combine the flexibility of a general-purpose processor with the efficiency of an application-specific hardware solution.

3.2.3 Co-development and Logic Design Methodology

The HNC node's design integrates hardware and software components forming a joint system in which software functions are reliant on hardware implementations and vice versa. This entails the concurrent development of hardware and software components in a co-development process. Naturally, this is accompanied by a corresponding co-design process. Additionally, verification here necessitates combined hardware-software testing, i.e., co-verification. The AMD Xilinx Vivado tools (AMD Xilinx, 2019e,b) used for development are designed to assist in these processes. The workflows here are complex and a comprehensive description would go beyond the scope of this thesis. Therefore, I will constrain the following to an explanation of the chosen approaches and highlight some aspects.

From the perspective of the top-level design, the co-development process conducted is a software-driven approach. However, there is a variety of constraints by the capabilities of the Zynq-7000 SoC device technology where hardware then becomes the driver for software development.

Throughout the development of the HNC node, co-verification became an important concern. The HNC node's software components have been written in the C language, while the majority

of the hardware design was carried out on the Register Transfer Level (RTL) using the VHDL[7] language – a rather error prone approach, which requires extensive testing and functional verification. A comprehensive description of the verification strategy, which introduces a rather unusual software-driven hardware verification approach leveraging the Zynq SoC architecture, is given in Chapter 4.

Choice of design entry – RTL vs. HLS

The decision to use the more laborious and time-consuming RTL design approach instead of High-Level Synthesis (HLS) (AMD Xilinx, 2019d) is motivated by the endeavor to maximize control over microarchitectural details to optimize timing behavior and achieve best possible performance – a key objective of the HNC node design. Attempts to use HLS for the design were not successful in this respect. The results did not meet expectations, neither in terms of the achieved latencies nor in terms of FPGA resource utilization.

The HLS design process takes a behavioral description formulated in a high-level programming language, such as C. From this, the design tools generate an RTL description guided by synthesis directives, i.e., code annotations that decorate the high-level description. It takes an understanding of how HLS works to find the right code style and directives so that the desired hardware architecture of an algorithm is reflected in the high-level description. At this level of abstraction, the details of the microarchitecture are almost inaccessible. Although HLS is an attractive methodology that reduces design times and also has matured the quality of results since introduced, it does not allow for the full exploitation of a design's optimization potential. Therefore, RTL-level block design has been chosen for hardware design entry.

Design flow

Development was conducted using the Integrated Development Environments (IDEs) provided with the Vivado tools. They bridge hardware and software development processes and provide the workflows for co-development.

The hardware blocks of the HNC node were developed using a mixture of RTL-level design and graphical block design. In this approach, RTL modules are integrated into the top-level design using the IP Integrator (AMD Xilinx, 2019c) of the Vivado Design Suite (AMD Xilinx, 2019e). The IP Integrator provides a graphical interface that allows the user to work interactively at the top level of the design. VHDL generics are exposed and can be modified to parameterize modules. This feature of the VHDL language was extensively used when exploring design alternatives.

[7] Very high speed integrated circuit Hardware Description Language

In order to interpret an IP from the processing system's perspective, hardware definitions are exported to the Vivado SDK (AMD Xilinx, 2019b) – the design entry for software development. This export generates a so called *Board Support Package* (BSP); a collection of software drivers customized to the provided hardware definitions, e.g., address mappings of PS-PL interfaces.

Hardware-software co-development has always been accompanied by a co-verification process (see Chapter 4 for further details).

3.3 HNC Node Architecture

3.3.1 Architecture Concept

Conceptually, the HNC node design is centered on FPGA-SoC technology aiming at a hybrid hardware-software architecture in which hardware and software components are tightly coupled. Here, the Zynq SoC device technology is exploited, specifically the ability to offload compute-intensive tasks to programmable logic for acceleration, combining the flexibility of a general-purpose processor with the efficiency of an application-specific hardware solution.

The underlying algorithms and the functional principle of the approach, however, do not differ from those that are typically used in pure software implementations for discrete-time simulations of spiking neural networks of point neuron models. It follows a hybrid strategy where neuron dynamics progress synchronously, time-driven and at fixed intervals, and synaptic inputs are processed asynchronously, event-driven, triggered when a presynaptic neuron emits a spike (see Morrison et al., 2005, and Section 1.2.3).

Figure 3.3 depicts the top-level architecture of the HNC node and illustrates the conceptual idea. Three main building blocks can be distinguished (from top to bottom): an off-chip external memory; an Application Processing Unit (APU); and a Programmable Logic (PL) part. Here, the APU is used to run a software system that prepares and supports the simulations and communicates with a Linux host system. This software system resides in the external memory. The PL implements the hardware blocks of a simulation engine.

Performance non-critical and critical tasks

From the perspective of the design goal of accelerated simulation, computational tasks can be divided into performance non-critical and performance critical tasks. The former would not benefit from hardware acceleration. It is therefore sufficient to implement these tasks in software and have them executed by a general-purpose processor. The flexibility that a software solution here offers is even an advantage and facilitates system integration, for example. Non-critical

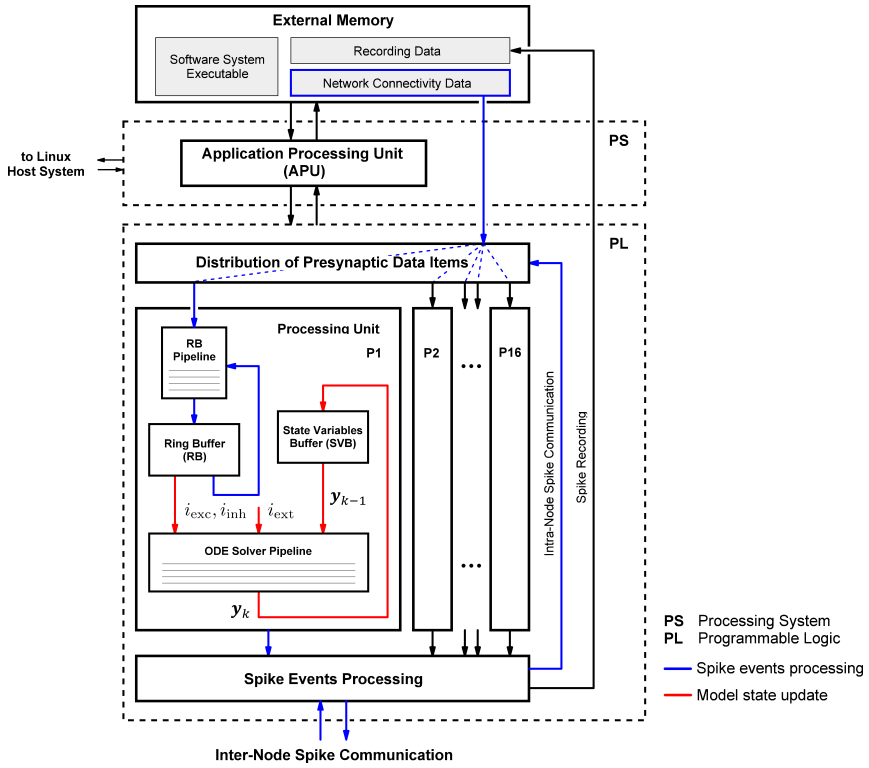


Figure 3.3 | HNC node architecture concept. At the top level, the HNC node architecture comprises three main building blocks (from top to bottom): an off-chip external memory; an Application Processing Unit (APU); and Programmable Logic (PL). The APU, as part of the Zynq device Processing Systems (PS), runs a software system. Its executable resides in external memory, which is also where the connectivity data of the network to be simulated is located. The actual simulation engine is implemented on the Programmable Logic (PL) section of the Zynq device. There are two distinct performance-critical tasks that are offloaded to the PL: the computations performed to advance model dynamics, the model state update (indicated by the red arrows); and the processing of spike events and synaptic inputs (indicated by the blue arrows).

are those tasks that are not directly related to the simulation itself. This applies to tasks such as: the instantiation of neurons and connections on a node; the control of the simulation; and the post-processing of simulation data. Tasks relevant to a node's simulation performance, and therefore offloaded to programmable logic, are: the computations performed to advance model dynamics (indicated by the red arrows in Figure 3.3); and the processing of spike events and synaptic inputs (indicated by the blue arrows in Figure 3.3). Since the HNC node is designed

to operate in a cluster, inter-node spike communication and synchronization between nodes are additional performance-relevant processes here.

Model state update – locality of data and operations

By implementing an algorithm in hardware locality of data and operations can be achieved, alleviating the inevitable limitations of traditional general-purpose architectures, i.e., the limitations imposed by the von Neumann bottleneck. Moreover, hardware is parallel by nature. Latency can be reduced and throughput increased by hardware-level parallelization. The HNC node implements 16 parallel processing units (P1 through P16) and also pipelines operations, where pipelining provides hardware parallelism at the operation level. This maximizes throughput; the work that is performed per unit time.

Each of the HNC node's processing units is capable of performing the computation of $N^{\mathrm{P}} = 64$ neurons. A HNC node is thus capable of simulating $N^{\mathrm{M}} = 1024$ neurons. The operations to calculate the dynamics of the N^{M} neurons are performed by the ODE solver pipelines, where the neuron's state vectors $\boldsymbol{y}_k$ are stored in the state variables buffers (SVBs). The data paths involved in this processing are indicated by the red arrows in Figure 3.3. Neuron states are updated at intervals of 0.1 ms – the temporal resolution of the simulation in the biological time domain. The SVBs are implemented as fast on-chip BRAM memories which creates the aforementioned data locality. However, technical constraints and functional requirements did not always allow for close proximity of data and operations in the HNC node design, as elaborated later in this section.

Network connectivity data and processing of synaptic inputs

The HNC node represents a network's connectivity as lists of synaptic connections, with one list maintained for each presynaptic neuron that has at least one postsynaptic neuron on the node. A list item contains information about the postsynaptic neuron, the synaptic weight, and the synaptic transmission delay of the connection. From this data the synaptic inputs to neurons are derived. These *lists of synaptic targets* form the *Network Connectivity Data*, which is stored in the off-chip external memory (top in Figure 3.3). This undermines the concept of data locality to some extent and is a decisive system performance limiting factor. There are two constraints that led to this design decision.

The first is a resource limitation. Storing the connectivity data in fast on-chip BRAM memories would be ideal, but BRAM is a limited FPGA resource and the memory requirement for storing a network's connectivity is demanding. For example, assuming a 64-bit data item to represent a synaptic connection, a network model, such as the cortical microcircuit model (Potjans and Diesmann, 2014), which comprises $0.8 \cdot 10^5$ neurons and $0.3 \cdot 10^9$ synapses, will require 2.4 GB

of memory in total. That is 24 MB per compute node if a single node processes 10^3 neurons. The XC7045 chip provides 19.2 Mbit of BRAM. This is ten times less memory than required.

The second constraint comes from a functional requirement. Although plasticity functionality was not in the focus of the design, in order to be able to cope with synaptic and structural plasticity algorithms in the future, network connectivity and the state of synapses need to be stored in an accessible and modifiable way. For static networks, performance efficient solutions have been developed that use procedural connectivity generation (see, e.g., Knight and Nowotny, 2021; Heittmann et al., 2022); instead of having to retrieve the synaptic connections from a memory, they are determined algorithmically during the runtime of a simulation. Due to the aforementioned requirement for plasticity functionality, such a solution was excluded and a compromise is made here in terms of achievable performance. The connectivity data is stored in external memory accessible by both the APU and the PL.

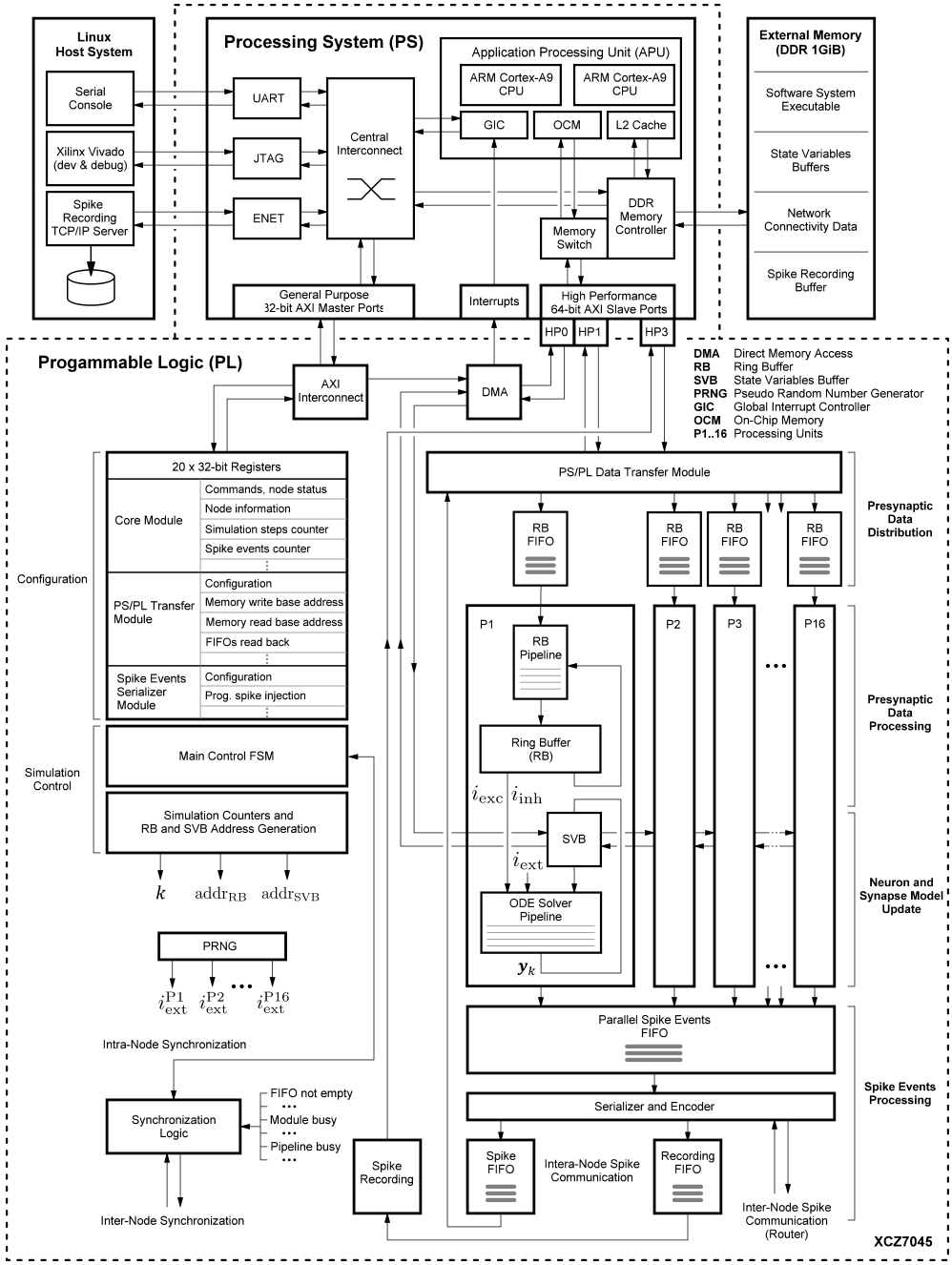
The HNC node parallelizes the processing of synaptic inputs. A received spike event triggers the retrieval of the list of synaptic targets of the presynaptic neuron that emitted the spike. This list is read from external memory and list items are distributed to the processing units (in Figure 3.3 indicated by the blue dashed lines). The data is passed through the ring buffer (RB) pipelines, which accumulate the synaptic inputs in the RB memories. The content of the RBs then constitutes the synaptic inputs (excitatory, i_{exc}, and inhibitory, i_{inh}) to the postsynaptic neurons.

The processing of postsynaptic spike events is less computationally intensive, but is also implemented on the PL to keep overall latencies low. The associated component is shown at the bottom of Figure 3.3, where it is labeled *Spike Events Processing*. The spike events are serialized and packed for communication and recording.

The conceptual ideas behind the HNC Node design can be summarized as follows: leveraging the flexibility of a general-purpose processor tightly coupled with programmable logic; offloading performance-critical tasks to programmable logic to establish data locality; parallelizing algorithms at the hardware level to reduce latency; and pipelining operations in hardware to increase throughput. Together, these principles enable efficient and high-performance processing.

3.3.2 System-Level Architecture

The previous section described the technical concept and explained the motivation behind some of the design choices. This section breaks this down further. Figure 3.4 shows the system-level view of the HNC node architecture design with the major building blocks, their interactions and functional assignments. The dashed frames enclose the on-chip components, PS (upper dashed frame) and PL (lower dashed frame). Central component of the PS is the Application Processing



Caption overleaf.

Figure 3.4 | System-level view of the HNC node hardware architecture. Shown are the main building blocks. The dashed frames enclose the on-chip components: Processing System (PS) (upper dashed frame); and the modules implemented in Programmable Logic (PL) (lower dashed frame). Attached to the PS is a 1GiB off-chip external memory (upper right). It stores the executable of the software system that orchestrates the operation of the HNC node. The external memory also stores the data structures required for operation, i.e., the state variables (a copy of the SVBs on the PL) and the network connectivity data, and is used to buffer the data recorded from simulations. The PS is connected to a Linux host system (upper left). The host system runs the development environment, a serial console, and a TCP/IP server to which the recorded simulation data is transferred. The functional assignment of the hardware blocks is marked on the right: presynaptic data distribution; presynaptic data processing; neuron and synapse model update; and spike events processing. For a description of the components, see the main text.

Unit (APU) which runs the HNC node software system. The simulation engine's core components are realized in programmable logic utilizing the PL section of the device. Functionally, hardware blocks can be assigned to the following four distinct tasks performed in a simulation step (marked on the right in Figure 3.4): (i) presynaptic data distribution; (ii) presynaptic data processing; (iii) neuron and synapse model update; and (iv) spike events processing.

Presynaptic data distribution: The main building block here is the *PS/PL Data Transfer Module*. Its operation is triggered by incoming spike events. Upon the occurrence of a spike event, the module initiates a sequence of read operations from external memory to obtain the node-local postsynaptic connections of the presynaptic neuron that has emitted the spike, i.e., the list of synaptic targets of the presynaptic neuron. This data is referred to here as the *presynaptic data*. To optimally use the available memory read bandwidth, the module is connected to the PS via a pair of high-performance ports. The presynaptic data items are then distributed to the 16 processing units according to where the corresponding postsynaptic neurons are being processed. To compensate for latencies, a series of first-in-first-out (FIFO) buffers (in Figure 3.4 labeled RB FIFO) connect the *PS/PL Data Transfer Module* to the processing units.

Presynaptic data processing: From the presynaptic data, the synaptic inputs are derived. The building blocks here are the ring buffers (RBs) and the ring buffer pipelines (RB pipelines) as part of the processing units. The presynaptic data items are fetched from the RB FIFOs and passed through the RB pipelines. The RB pipelines accumulate the presynaptic inputs, lumping together a neuron's excitatory synaptic inputs i_{exc}, and inhibitory synaptic inputs i_{inh}, respectively, and store the resulting values in the RBs. The RBs provide these *weighted* synaptic inputs to the ODE solver pipelines, while also realizing the synaptic transmission delays.

Neuron and synapse model update: The processing unit's ODE solver pipelines implement the functional blocks that compute the neuron and synapse model dynamics. They receive input

from the RBs and update the model state vectors $\boldsymbol{y}_k$, which are stored in the state variables buffers (SVBs). An additional external random input $\{i_{\text{ext}}^{\text{P1}}, .., i_{\text{ext}}^{\text{P16}}\}$ to neurons can be provided by a pseudo-random number generator (PRNG). When a neuron emits a spike, the event is pushed into the *Parallel Spike Events FIFO* (shown at the bottom of Figure 3.4). Here, due to the parallel operation of the processing units, also multiple spike events can occur in the same clock cycle.

Spike events processing: Spike events are serialized and packed for local (intra-node) and global (inter-node) spike communication as well as for recording. The main building block here is the *Serializer and Encoder Module*. It fetches the spike events from the *Parallel Spike Events FIFO*, processes the events, and stores the encoded events in two FIFO buffers, the *Spike FIFO* and the *Recording FIFO*. The *Spike FIFO* compensates the latencies of the intra-node spike communication. The *Recording FIFO* buffers the spike recordings. It is read asynchronously by the *Spike Recording Module*, which writes the data to the external memory, where the recordings are buffered again before being transferred to the Linux host system (shown at the upper left in Figure 3.4).

To orchestrate the above processes, additional components are required to manage control flows, configure modules, enable the APU to access on-chip resources, and synchronize processes.

A set of 32-bit registers store configuration and status information (shown in the mid left of Figure 3.4; a description of the registers can be found in Appendix **C**). Their settings steer the operation of a finite state machine (FSM) (labeled *Main Control FSM* in Figure 3.4). This FSM generates the control signal sequences for the different HNC node operations, such as the loading and unloading of state variables and the execution of a simulation. The registers are mapped into the APU's address space and are thus accessible by the HNC node software system.

In order for the APU to perform software-controlled read and write operations on memories on the PL, in particular to access SVBs for loading and unloading state variables, all processing units are chained to one another and connected to a direct memory access (DMA) controller. This enables efficient data exchange between the APU and the processing units.

To guarantee the temporal causality of spike events and a coherently evolution of algorithmic time in a cluster, a node-internal (intra-node) synchronization and a synchronization between nodes (inter-node) is required. For this purpose a synchronization logic (shown at the lower left in Figure 3.4) monitors the operating status of all modules as well as the status of the cluster. Technically, it implements a barrier mechanism that synchronizes the overall processing at the end of each simulation time step.

3.3.3 Software System

The entire function of the HNC node is controlled by its software system. In order to minimize the resources footprint and achieve the best possible performance, its implementation was done in the C language as a bare-metal application; i.e., the executable runs natively on the APU without the use of an underlying operating system.

3.3.3.1 Software System Architecture

The HNC node software system design uses a tiered software architecture style. Figure 3.5 outlines the architecture. At the lowest level, a hardware abstraction layer (Figure 3.5A) provides basic routines fundamental to drive the functions of hardware components, and hides the technical details of hardware implementations. This abstraction layer implements the functions for initializing the hardware blocks on the PL, establishing a basic serial console and TCP/IP communication with the host system, performing PS-PL DMA transfers, and handling interrupts. A layer of service routines (Figure 3.5B) builds on top of the hardware abstraction layer providing system low-level functionality. It implements, for example, an interface that provides convenient access to the processing units on the PL for the loading and unloading of state variables. This interface consists of a DMA driver wrapper that is aware of the processing unit's microarchitecture. Other service functions that the layer implements provide routines for data type conversion and the assignment of neurons to hardware resources. These services are called by higher-level simulator functions, such as the functions of the *Neuron Manager*, the *Connection Manager*, the *Recording Client* (Figure 3.5C), and the functions that control node operation (Figure 3.5D).

Neuron Manager, *Connection Manager*, and *Recording Client* constitute the central part of the HNC node software system. They serve the following purposes:

The ***Neuron Manager*** is responsible for the instantiation of neurons. It invokes lower-level functions to assign a neuron a processing unit and a relative position in the sequence of the ODE solver pipeline processing. It also initializes the associated SVB memory structures and sets up neuron parameters and initial states.

The ***Connection Manager*** creates the synaptic connections, which amounts to creating the network's connectivity data and setting up lists of synaptic targets in memory.

The ***Recording Client*** reads the buffered spike recordings from external memory and transfers the packed data to a TCP/IP server that runs on the Linux host system.

At the highest level (Figure 3.5E), a C-API provides three function calls – the minimum required to instantiate and simulate a network. These function calls are: `Create(..)`; `Connect(..)`;

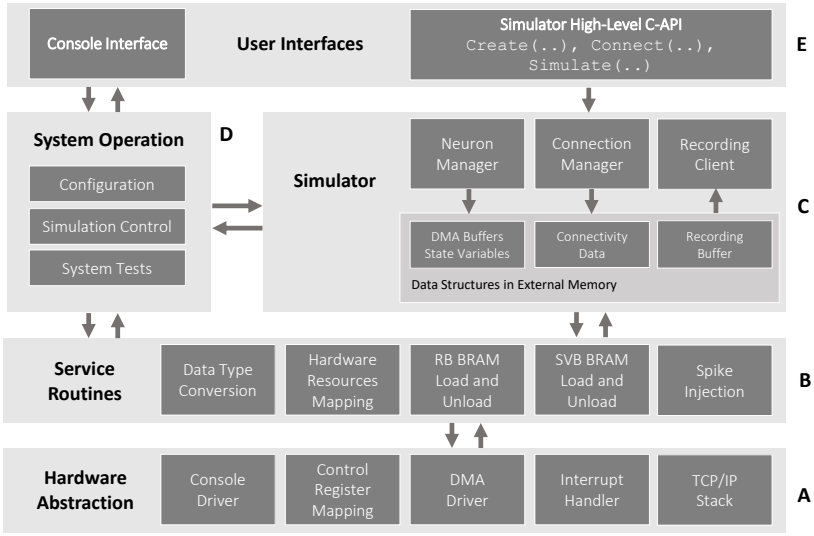


Figure 3.5 | HNC node software system architecture. The tiered architecture provides abstraction at different functional levels. At the lowest level, a hardware abstraction layer (**A**) hides technical details about the implemented hardware components. A set of low-level routines (**B**) builds directly on the hardware abstraction layer and provides service routines for the higher-level modules and functions of the *Neuron Manager*, the *Connection Manager*, the *Recording Client* (**C**), and for system operation (**D**). At the highest level (**E**), a C-API exposes functions to instantiate and simulate a network. For user interaction, a command-line console interface is provided. The arrows indicate the dataflow between modules – they do not reflect the function call hierarchy, which is strictly from higher-level layers to lower-level layers.

and `Simulate(..)`. For user interaction, also a command-line console interface is provided.

The HNC node software system is completed by a set of supervisory routines (Figure 3.5D) that implement functions for the user-interactive control of node operation and to support system testing and debugging.

3.3.3.2 Technical Concepts

There is a variety of technical details in the HNC node implementation concepts intended to foster efficient resource management and resource utilization. Four of these underlying technical concepts are briefly explained in the following, as they are necessary for further understanding.

Node-local neuron-id

To identify a neuron in a network, neurons are typically numbered sequentially, with each neuron being assigned a unique id, a neuron-id. In a cluster of nodes, a neuron-id can be decomposed into

a node-id and a node-locally unique identifier. The HNC node uses such a node-local identifier; here referred to as the *local neuron-id* of a neuron denoted as n_{loc}.

Hardware resources identifier

The hardware resources with which a neuron gets associated on the PL are identified by a *hardware resources identifier* (HwResId). A HwResId is represented by a data structure that stores information about the assigned processing unit, the neuron's position in the sequence of the ODE solver pipeline processing as well as information about the associated PS-PL data path and the related RB FIFO buffers. The service routines layer provides the functions to assign these resources, mapping a local neuron-id to a HwResId, where a HwResId is always unique. For an instantiated network and during a simulation, the assignment is static, and exactly one HwResId per local neuron-id exists.

State variables and data type conversion

There are two types of SVB resources: the hardware SVBs of the processing units on the PL, and a copy of it in external memory maintained by the *Neuron Manager*; in the following referred to as SVB^{NM}. Neuron states are stored in the SVBs as generic 128-bit data words (see also Section 3.3.4.4). The exact data format depends on the model-specific implementation of the ODE solver pipelines. Setting up state variables therefore requires a model-specific data type and endianness conversion. The APU's ARM processor cores use Little-Endian and provide 32-bit floating point single precision. The data paths on the PL are laid out in Big-Endian byte order. Depending on the model's demands in terms of required numerical precision, different data types may be needed. In this respect, the hybrid architecture of the HNC node provides a level of flexibility that allows any user-defined data type to be implemented.

Presynaptic data representation – network connectivity data

The HNC node represents connections (synapses) on the same node as their postsynaptic neuron, where connections are stored as lists of quadruples, referred to here as *lists of synaptic targets*

$$\text{LST}_j = \{(n_{\text{loc},i}, w_{ij}, d_{ij}, s_i), .., ()\}. \tag{3.1}$$

In Equation (3.1), j identifies the presynaptic neuron to which an LST belongs, $n_{\text{loc},i}$ specifies the node-local id of a postsynaptic target neuron i, w_{ij} and d_{ij} denote the synaptic weight and delay of a connection, and s_i is a hardware control value derived from the postsynaptic neuron's HwResId. An illustration of a neuron's connections is shown in Figure 3.6.

These LSTs are maintained by the *Connection Manager*, where for each presynaptic neuron j that has at least one postsynaptic target on the node a list LST_j is being held. In this concept,

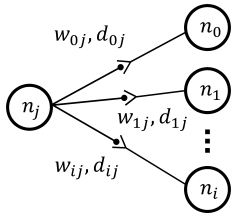


Figure 3.6 | Synaptic target connections. Synaptic connections of a presynaptic neuron n_j to its postsynaptic targets $\{n_0, n_1, .., n_i\}$.

global connectivity is implicitly encoded by the location of an LST in memory. A single element within an LST is represented in memory by a 64-bit data structure. The data format is outlined in Appendix **D**. This data structure is determined by the microarchitecture of the *PS/PL Data Transfer Module* (see Section 3.3.4.2), as well as the microarchitecture of the RBs and RB pipelines (see Section 3.3.4.3). These hardware blocks process the elements of an LST and perform the calculations using a 40-bit fixed-point data type for synaptic weights and a 9-bit unsigned integer data type for synaptic delays. All LSTs of all nodes together represent a network's connectivity data. To distinguish between the *raw* data stored in the LSTs and the processed *weighted synaptic inputs* to a neuron, the data stored in the LSTs is here also referred to as the *presynaptic data*, and an LST element is referred to as *presynaptic data item*.

The current HNC node prototype is capable of simulating 1024 neurons per node, where a presynaptic neuron can connect with up to 128 postsynaptic neurons on a node. This is a sufficient number of connections, as it aligns with the connection probability value of approximately $\epsilon = 0.1$ observed by Braitenberg and Schüz (1998) in cortical tissue. It holds

$$C^{\mathrm{M}}_{\mathrm{max}} > \epsilon N^{\mathrm{M}}, \tag{3.2}$$

where $C^{\mathrm{M}}_{\mathrm{max}}$ denotes the maximum number of postsynaptic target connections per presynaptic neuron and node, and N^{M} is the number of neurons per node; a value of $C^{\mathrm{M}}_{\mathrm{max}} > 102$ is therefore sufficient here.

A typical cortical neuron connects to between 10^3 and 10^4 other neurons. A network of $N = 10^5$ neurons, with each neuron forming 10^4 connections, thus represents an upper limit on the number of synapses per node. For network sizes $N > 10^5$, the total number of synapses in a network scales linearly with network size; below this size, it scales quadratically. The amount of

memory required on a node to store a network's connectivity data is then given by

$$S^{\mathrm{M}}_{\mathrm{conn}} = \begin{cases} \epsilon N^{\mathrm{M}} N w_{\mathrm{len,LST}} & \text{if } N \leq 10^5 \\ \dfrac{10^4}{M} N w_{\mathrm{len,LST}} & \text{otherwise,} \end{cases} \tag{3.3}$$

where $w_{\mathrm{len,LST}}$ is the word length of an LST element (here, 64 bit), and M specifies the number of nodes required to simulate a network of size N for a given number of neurons per node N^{M}. To give an an order of magnitude for the memory requirement $S^{\mathrm{M}}_{\mathrm{conn}}$, depending on the number of neurons simulated per node, a large-scale network with a few million neurons would require approximately several hundred megabytes of memory per node, assuming each connection is represented by a 64-bit data word.

3.3.3.3 Node-Local Network Instantiation

A network is instantiated on the HNC node by a sequence of C-API `Create` and `Connect` function calls. These are processed by the *Neuron Manager* and the *Connection Manager*, respectively. A `Create` call instantiates a single neuron. A `Connect` call creates a single synaptic connection. Both the instantiation of a neuron and the creation of a synaptic connection require the allocation of various system resources. These resources are maintained by the *Neuron Manager* and the *Connection Manager*, as described in the previous section.

`Create`: The `Create` call takes as its arguments a neuron model type, a local neuron-id, and the initial values of the neuron's state variables. The call is processed by the *Neuron Manager*. The sequence diagram in Figure 3.7A shows the interaction between the software layers and functions invoked. The processing of a `Create` call comprises two steps. Firstly, the *Neuron Manager* invokes a service routine to locate the position of the neuron's state variables in the $\mathrm{SVB}^{\mathrm{NM}}$ in external memory. This position is derived from the HwResId assigned to the neuron's local neuron-id. Secondly, the *Neuron Manager* builds the data structure in memory setting up model parameters and initializing state variables, i.e., setting the initial state of the neuron. This includes the conversion of data types into the data formats and endianness used by the neuron and synapse model-specific hardware blocks of the ODE solver pipelines.

`Connect`: The `Connect` call expects in its argument list a local neuron-id of a presynaptic neuron (in a multi-node system this is extended by the node-id), a local neuron-id of a postsynaptic neuron, a synaptic weight, and a synaptic delay value. The call is processed by the *Connection Manager*. Figure 3.7B shows the sequence diagram. The processing of a `Connect` call also requires two steps. First, the HwResId of the presynaptic neuron and the HwResId of the postsynaptic neuron

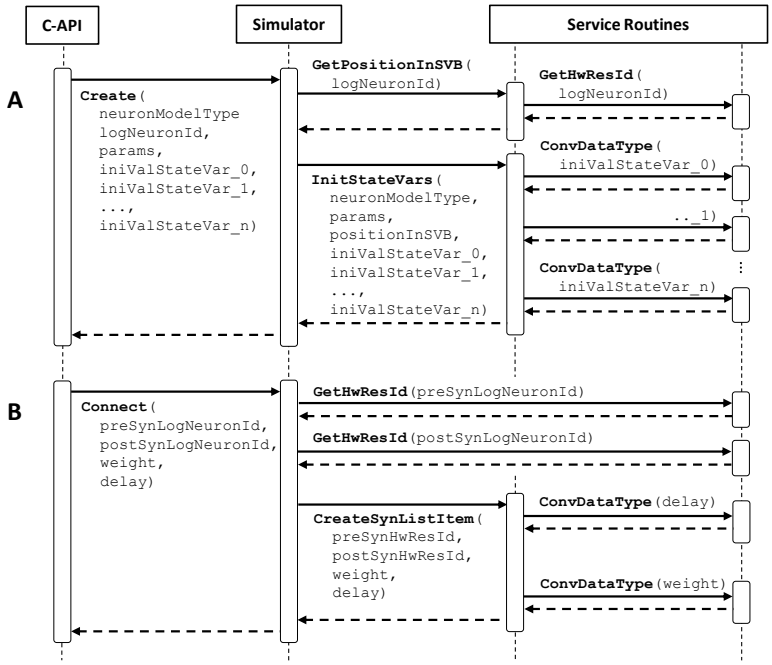


Figure 3.7 | Sequence diagram. Interaction of software layers and functions invoked by the *Neuron Manager* and the *Connection Manager* when processing: (**A**) a `Create`; and (**B**) a `Connect` C-API function call.

are obtained by invoking a service routine. In a second step, an LST element $(n_{\text{loc},i}, w_{ij}, d_{ij}, s_i)$ is created and appended to the presynaptic neuron's list of node-local target connections, LST_j. Data type conversion and managing byte order are also necessary here; weights are specified in 32-bit floating point but stored in RBs and processed by the RB pipelines using a 40-bit fixed-point format.

Note that the function calls `Connect` and `Create` do not require any component on the PL part of the HNC node. They are executed solely by the APU.

3.3.3.4 Simulation

A simulation is launched with a `Simulate` function call. It prepares the processing units on the PL and triggers the hardware blocks to run the simulation.

`Simulate`: The `Simulate` function call takes as its only argument the duration for which a

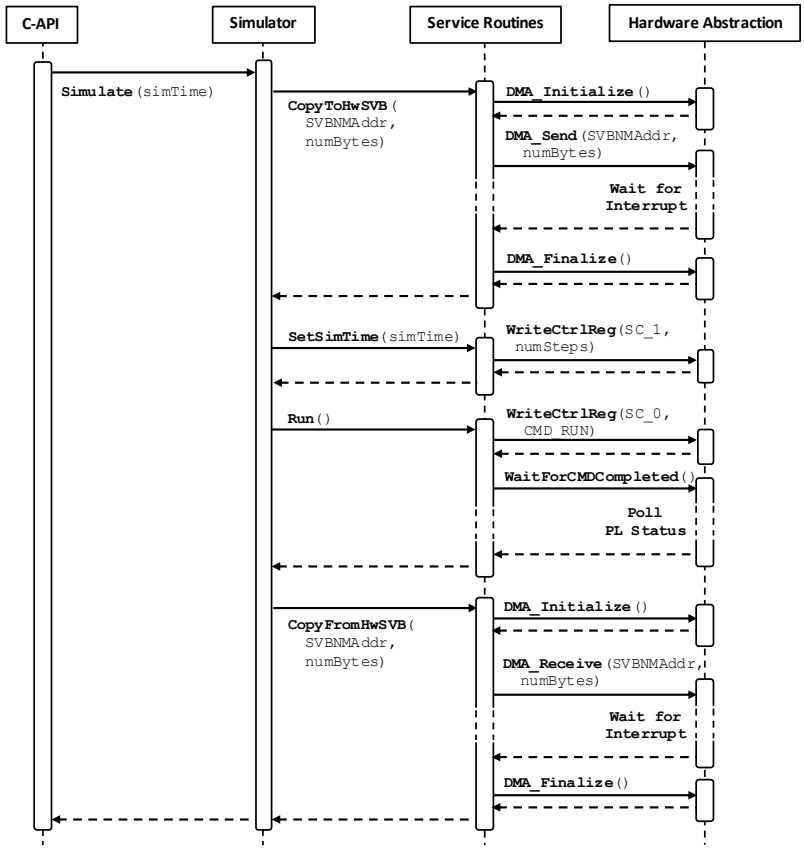


Figure 3.8 | Sequence diagram. Shown is the interaction of software layers and the functions invoked when processing the `Simulate` C-API function call.

network is to be simulated, i.e., the *simulated time*. The execution of the `Simulate` function call requires the simulation layer to perform four steps, shown in the sequence diagram in Figure 3.8. These steps are: copy the state variables from the SVB^{NM} in external memory to the processing unit's SVBs on the PL; set the simulated time; trigger the simulation run, i.e., the main FSM on the PL; and finally, when the simulation has completed, copy the state variables back from the SVBs to the SVB^{NM}. Setting the simulated time and starting a simulation are simple write operations to control registers (a description of the registers is provided in Appendix **C**). More complex is the transfer of the state variables between the SVB^{NM} and the processing unit's SVBs.

Here, a DMA controller is used for which the service routines layer provides an interface that encapsulates its function. The DMA functions are aware of the SVB microarchitecture, as does the *Neuron Manager*, which maintains the SVB data structures.

The current implementation of the HNC node also provides functions to advance a simulation for a specified time interval, halt, and resume it later as well as to perform a simulation in a step-wise manner. Also implemented is an operation mode that allows the processing units on the PL to be operated in closed-loop with the APU (see Section 3.3.4.4.1). From the perspective of the software system, these functions are just variations of the sequence of operations of the `Simulate` call shown in Figure 3.8.

There are two technical aspects related to performance worth mentioning here: DMA latency; and the use of the external memory by the APU and the hardware blocks on the PL.

DMA latency: In the current implementation, an AMD Xilinx AXI DMA soft-IP core is used (AMD Xilinx, 2019a). The DMA transfer of the state variables takes approximately $30\mu s$[8], transferring 16 KiB (1024 x 128bit) of data, where the DMA controller raises an interrupt request notifying the APU when a transfer is completed. This latency is relevant for APU closed-loop operation and when recording state variables (see next section).

Use of external memory: During simulation, the APU does not access external memory, allowing the components on the PL to make optimal use of the available memory read bandwidth when processing incoming spike events.

3.3.3.5 Recording

In simulations of spiking neural networks, the data of interest are the recordings of the neurons' spike trains, but also the evolution of the model dynamics (e.g., membrane potentials) is data that is often subject to analysis. The HNC node allows recording of both the spike trains of neurons and their state variables.

Spike events

The recording of spike events is a fully asynchronous process that runs in parallel and decoupled from simulation. During a simulation, spike events are grouped together and packed into 64-bit binary values. These are buffered in the *Recording FIFO* (shown at the bottom in Figure 3.4) before being written to external memory. In external memory, the HNC node maintains a 60 MiB recording buffer capable of caching approximately 15 million spike events. The buffer is written in a round-robin fashion by the *Spike Recording Module* which uses a high-performance port for

[8] The value was measured on the APU using a hardware timer.

the transfer (the write data channel of HP3). After a simulation has completed, the *Recording Client* (Figure 3.5C) streams the data to a TCP/IP server running on the Linux host system, where the spike events are unpacked. To establish the TCP communication, the *Recording Client* uses the open-source *lightweight IP*[9] (lwIP) TCP/IP stack that comes with the AMD Xilinx board support package included in the Vivado SDK.

Parallel access to the external memory when writing spike recordings and reading presynaptic data simultaneously carries the risk of memory read-write contention. This has been investigated by comparing the runtimes of simulations when spike recording is turned on and when it is turned off. Even when all neurons were forced to exhibit very high spike rates, no major differences in runtimes were observed.

State variables

State variables are stored in the processing unit's SVBs. For recording, the entire contents of the SVBs are copied to external memory. The copy operation is performed using a DMA transfer, which requires a simulation to be paused. Consequently, recording neuron states significantly slows down a simulation – as stated earlier, a DMA operation takes approximately 30 μs. Nevertheless, the DMA transfer provides an efficient method to capture all state variables at once and at any desired interval.

3.3.3.6 Interactive User Console Interface

The HNC node provides a serial console communication interface. It is established at startup, where it is used to log the startup sequence. During this sequence, the HNC node software system performs a series of internal tests, initializes data structures, and sets the hardware blocks on the PL to a defined state. The console interface also allows the user to work interactively with the system. It provides low-level access for system administration, allowing, for example, status and configuration information to be queried and system operation to be monitored. For this purpose, the software system provides a number of functions such as: reset and initialize the HNC node; view configuration and status registers; set configurations; print RB and SVB memories; perform functional tests; create neurons and connections; look up connections, including a reverse look-up; load a network from a set of configuration parameters; and run a simulation.

3.3.4 Hardware Microarchitecture

The HNC node software system interacts with the simulator hardware blocks and controls all node operations. There is a tight coupling between the two, not only in terms of the physical

[9] http://savannah.nongnu.org/projects/lwip/

coupling of PS and PL on the SoC device, but also logically at the level of data structures and functionality. This is reflected in the design of the node software as well as in the hardware microarchitecture; data structures are determined by microarchitecture details, and hardware blocks serve software functions.

The following sections describe the functional principles and microarchitecture of the most relevant hardware blocks and their interrelationships. Design decisions and their motivation are discussed, and where of thematic interest, designs are evaluated and architecture alternatives are presented. The order in which the hardware blocks are presented follows the order of their functional assignment to the processes of: presynaptic data distribution; presynaptic data processing; neuron and synapse model update; and spike events processing (see Section 3.3.2). Also described are the generation of random numbers and the synchronization processes that the HNC node performs at the end of each simulation time step.

3.3.4.1 Technical Concepts

Analogous to the description of the concepts underlying the software system, an insight into the technical principles and concepts underlying the hardware architecture is provided below. The functional principle of a ring buffer, the byte order used by hardware blocks on the PL, and some design decisions regarding the choice of data types are explained.

Ring buffer working principle

A ring buffer (RB) stores the weighted synaptic inputs of a set of postsynaptic neurons and also realizes the synaptic transmission delays. For this purpose, an RB implements multiple circular memory structures, the circular buffers. A neuron is assigned two of these structures, each of which is used to lump together a neuron's excitatory and inhibitory inputs, respectively (see Section 1.2.2 for an explanation). An RB thus consists of $2N^{\mathrm{RB}}$ circular buffers, where N^{RB} denotes the number of neurons it serves. The working principle is illustrated in Figure 3.9, where Figure 3.9A shows only one of the two memory structures assigned to a neuron; to both excitatory and inhibitory inputs, the same working principle applies. An RB is divided into *segments* (Figure 3.9B). A single segment stores the weighted synaptic inputs of N^{RB} neurons that is valid in a particular time step k.

When a presynaptic data item is received – henceforth this process is referred to as a *synaptic event* – it conveys information about the postsynaptic neuron, the synaptic weight, and the transmission delay of the connection, held in an LST. In the following, synaptic events will be denoted as (N, J, D), where N specifies the postsynaptic neuron, J is the synaptic weight, and D

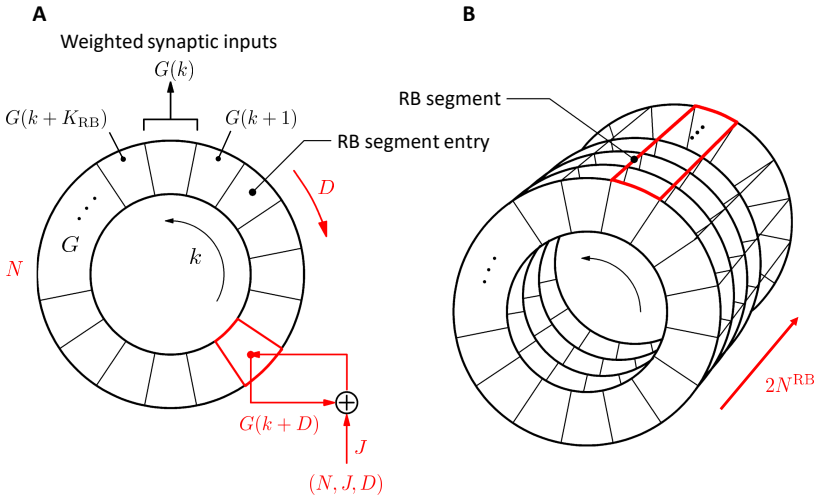


Figure 3.9 | Ring buffer working principle. (**A**) A circular buffer structure is employed to accumulate the synaptic inputs J of a postsynaptic neuron N and to delay their *delivery* according to the synaptic transmission delays D. Each neuron is assigned two circular buffers, one for excitatory inputs and one for inhibitory inputs (only one is shown). (**B**) A ring buffer (RB) consists of $2N^{\mathrm{RB}}$ circular buffers and stores the weighted synaptic inputs of N^{RB} neurons. An RB is divided into segments. When the simulation time is advanced to the next time step $k+1$, the structure is rotated by one segment, so that the synaptic input $G(k)$ of a neuron always corresponds to the simulation time.

is the transmission delay of the connection.

When a synaptic event is processed, first, the circular buffer that corresponds to the postsynaptic neuron N is selected, and from the current simulation time step k and the synaptic delay value D an offset into this buffer is calculated as $k+D$. This determines the RB *segment entry* $G(N,k+D)$ that is associated with the synaptic event. Second, the content of this segment entry is taken and the synaptic weight J is added to it. This is repeated for all synaptic events received in a simulation time step k. Finally, when the simulation time is advanced to time step $k+1$, the structure is rotated by one segment (i.e., $G(k) \leftarrow G(k+1)$, $G(k+1) \leftarrow G(k+2)$, ...), so that the synaptic input $G(k)$ of a neuron always corresponds to the simulation time. In this manner, synaptic inputs are accumulated and their *delivery* is delayed according to the synaptic transmission delays.

After a complete rotation of the structure, segment entries are reused. The number of segments K_{RB} thus defines the maximum possible synaptic delay, which is given by $D_{\mathrm{max}} = h(K_{\mathrm{RB}}-1)$, where h is the temporal resolution of the simulation.

The concept of accumulating a neuron's synaptic inputs using a circular buffer structure is

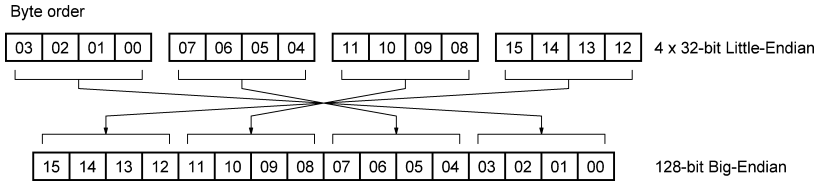


Figure 3.10 | Byte order. Little-Endian and Big-Endian byte order for a 128-bit compound data type.

well known from software simulators, such as the neural simulation tool NEST (Gewaltig and Diesmann, 2007; Morrison et al., 2005). This approach provides an efficient mechanism to process synaptic events. Nonetheless, its algorithmic implementation on von Neumann architectures is a non-trivial exercise, and has significant impact on the overall system performance. In simulations of large-scale networks, a compute node has to perform thousands of RB updates per simulation time step. The irregular memory access patterns lower a CPU's cache hit ratio, leading to additional memory access latencies and thus impeding efficient execution (see, e.g., Pronold et al., 2022). The hardware blocks of the HNC node that are associated with presynaptic data distribution and processing are therefore key components with a high impact on system performance.

Byte order

The 32-bit ARM processor cores of the APU use Little-Endian byte order. For the hardware blocks on the PL, it is technically more convenient to operate with Big-Endian. The use of Big-Endian has an advantage especially when user-specific data types or compound data types[10] cross 32-bit word boundaries. In Big-Endian, bits are always sorted according to their significance, which simplifies hardware design. Figure 3.10 shows an example of the reordering of the 32-bit data words for a 128-bit compound data type. Such a data type is used to store the state variables in the SVBs on the PL. The necessary endianness and type conversion can be done conveniently in software, and only needs to be done once when instantiating a network.

Data types

Flexibility in model implementation, numerical accuracy, and the reproducibility of simulation results are key requirements. This led to three design decisions regarding the data types used in hardware blocks: (i) for state vectors, a generic 128-bit data word is used; (ii) synaptic delay values are represented as unsigned integers; and (iii) for synaptic weights and synaptic inputs an

[10] A compound data type is a data type that consists of more than one element, where elements can be of any other data type.

s16.23 representation, i.e., a 40-bit fixed-point data type, is used. The reasons for these design choices are as follows.

Generic 128-bit state vectors: The use of a generic data type aims at flexibility in the model implementation. The exact format of the 128-bit word is defined by the model-specific ODE solver pipeline implementation and its software counterparts, i.e., the corresponding *Neuron Manager* functions. This concept allows flexibility in the number of state variables, their data types, and the numerical precision required by the model. This also allows for mixed-precision.

Unsigned integer delay values: In order to avoid rounding errors for synaptic delay values, APU and hardware blocks store them as unsigned integers (32-bit on the APU and 9-bit on the hardware blocks), where the value represented by the least significant bit corresponds to a delay value of 0.1 ms – the temporal resolution at which the HNC node performs simulations.

Fixed-point synaptic weight values in s16.23: The use of signed fixed-point numbers for synaptic weights aims at the replicability of simulation results. Generally, floating-point operations deliver results with a rounding error that usually cannot be guaranteed to be strictly less than the value represented by the least significant bit if it is 1; the unit in the last place (ulp). As a consequence, addition and multiplication become non-commutative operations. In simulations, a changed order in the sequence of spike events can therefore potentially result in differences in the accumulated synaptic inputs. In this respect, inadequate numerical accuracy can hinder replicability, that is, in repeated simulations, the results are no longer spike-for-spike identical. When using fixed-point numbers for the representation of synaptic weights, this can be avoided. Their addition is commutative. A change in the order of the spike events will not have any effect on the synaptic inputs. The use of the 40-bit s16.23 data type is motivated by the results obtained from the calculation verification tasks described in Section 2.3.4.2. This data type provides the numerical precision and value range that is needed to achieve sufficient simulation accuracy.

3.3.4.2 Presynaptic Data Distribution – PS/PL Data Transfer Module

The purpose of the *PS/PL Data Transfer Module* (DTM) is to efficiently retrieve the presynaptic data from the external memory and distribute it to the processing units. The microarchitecture of the module and how it interacts with the PS and the external memory is shown in Figure 3.11. The components of the DTM are framed by the dashed line.

To optimally utilize the read bandwidth of the external memory, the module bundles two high-performance ports to connect to the PS. For this purpose, it implements two 64-bit AXI master stream interfaces. The two high-performance ports have been selected such that they can operate in parallel and independently of one another; here HP1 and HP3 are used. The use of

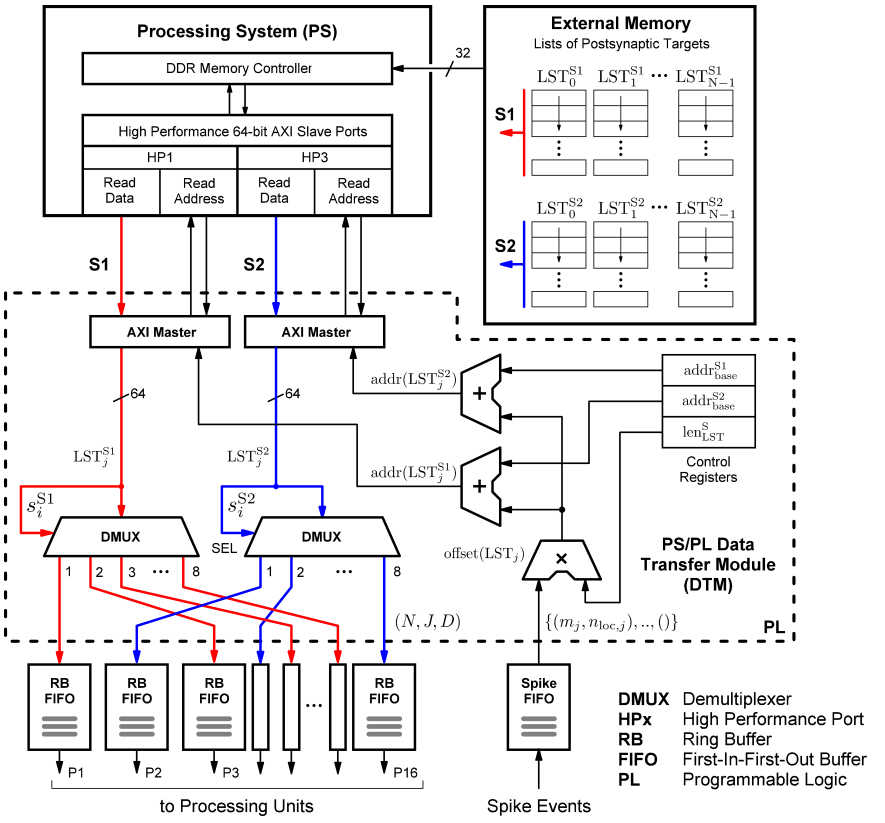


Figure 3.11 | PS/PL Data Transfer Module microarchitecture and interaction of components. The purpose of the *PS/PL Data Transfer Module* (framed by the dashed line) is to efficiently retrieve the presynaptic data from the external memory (upper right). Arriving spike events are first queued in the *Spike FIFO* buffer (lower right) and then processed sequentially. Each spike event triggers a sequence of read operations from external memory. To access the external memory, the module implements two 64-bit AXI master stream interfaces. These are connected to two high-performance ports, HP1 and HP3. This allows two independent data streams to be created and the available memory bandwidth to be optimally utilized. The two data streams are indicated by the red and blue arrows labeled S1 and S2. A presynaptic neuron's list of postsynaptic targets is split in memory into two lists, LST_j^{S1} and LST_j^{S2}, each associated with one of the two streams (upper right). Two demultiplexer (DMUX) circuits control the data paths of the streams and distribute the LST elements to the processing units. See also the description in the main text.

two ports instead of one allows data to be retrieved as two independent streams, maximizing memory read efficiency. The two streams are indicated in Figure 3.11 by the red and blue arrows

labeled with S1 and S2. This design splits a presynaptic neuron's list of postsynaptic targets, LST_j (see Section 3.3.3.2), into the two lists $\mathrm{LST}_j^{\mathrm{S1}}$ and $\mathrm{LST}_j^{\mathrm{S2}}$, each assigned to one of the two data streams (shown in the upper right)

Arriving spike events are first queued in the *Spike FIFO* buffer (shown in the lower right of Figure 3.11). They are processed sequentially. Each spike event triggers the AXI master interfaces to perform a sequence of read operations, which retrieve the lists $\mathrm{LST}_j^{\mathrm{S1}}$ and $\mathrm{LST}_j^{\mathrm{S2}}$ from external memory in parallel. Note that accessing external memory here does not require the APU. The addresses at which the two lists are located in memory are calculated in two steps.

First, from the presynaptic neuron's node-id m_j and local neuron-id $n_{\mathrm{loc},j}$ a relative offset position in memory is derived as

$$\mathrm{offset}(\mathrm{LST}_j) \leftarrow \mathrm{offset}(m_j, n_{\mathrm{loc},j})\ \mathrm{len}_{\mathrm{LST}}^{\mathrm{S}}, \tag{3.4}$$

where $\mathrm{len}_{\mathrm{LST}}^{\mathrm{S}}$ denotes the length of a list associated with a stream. In a second step, this offset is then added to the memory base addresses of the two streams as

$$\mathrm{addr}(\mathrm{LST}_j^{\mathrm{Sx}}) \leftarrow \mathrm{addr}_{\mathrm{base}}^{\mathrm{Sx}} + \mathrm{offset}(\mathrm{LST}_j)\ \ \forall\ \mathrm{Sx} \in \{\mathrm{S1}, \mathrm{S2}\}, \tag{3.5}$$

where $\mathrm{addr}_{\mathrm{base}}^{\mathrm{Sx}}$ are the addresses at which the first list associated with S1 and S2, respectively, in memory begins. In the prototypical implementation, the lists are padded to have the fixed length

$$\mathrm{len}_{\mathrm{LST}}^{\mathrm{S}} = \frac{1}{2}\, w_{\mathrm{len,LST}}\, C_{\mathrm{max}}^{\mathrm{M}}, \tag{3.6}$$

where $C_{\mathrm{max}}^{\mathrm{M}}$ is the maximum number of a presynaptic neuron's postsynaptic targets on a node, and $w_{\mathrm{len,LST}}$ is the word length of an LST element. With $C_{\mathrm{max}}^{\mathrm{M}} = 128$ and $w_{\mathrm{len,LST}} = 64$ bit (see Section 3.3.3.2), $\mathrm{len}_{\mathrm{LST}}^{\mathrm{S}}$ results in 512 bytes. Thus, for each spike event, the DTM retrieves 1 KiB from external memory.

The memory base addresses of the two streams and the AXI transmission protocol properties are module configuration parameters. They are stored in control registers (see Appendix **C**). Their values are set during HNC node initialization. This allows (the software system) to relocate the data in memory and also to adjust the list length if necessary.

Each LST element also carries a 4-bit data path control value, s_i^{Sx}. It encodes the processing unit to which a data item is to be distributed, according to the assignment of the postsynaptic neuron to a processing unit (the encoding of s_i^{Sx} can be found in Appendix **D**). The value controls the demultiplexer (DMUX) circuits which set the data paths. The demultiplexers alternate in connecting the data paths to the RB FIFO buffers, which are attached to the processing units.

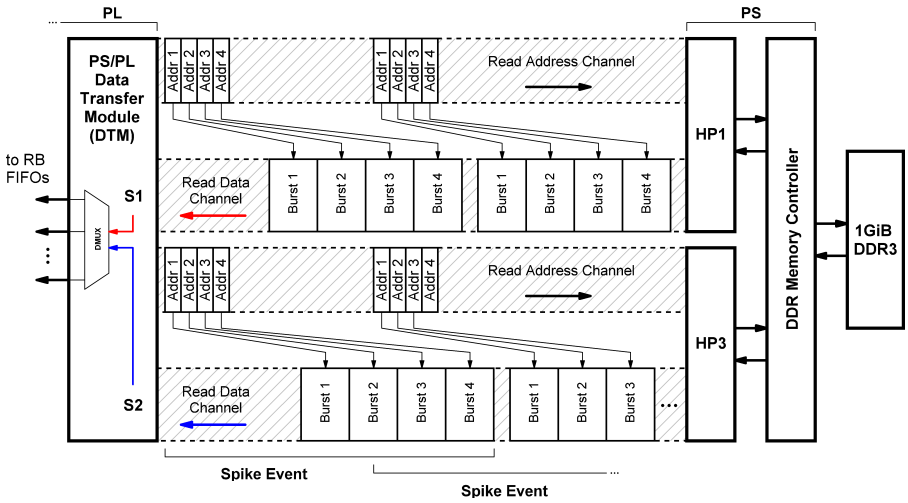


Figure 3.12 | AXI stream protocol implementation. The implemented protocol bundles two high-performance AXI stream port interfaces, HP1 and HP3. Each incoming spike event initiates the transfer of a sequence of four data bursts on each of the two read data channels, where a data burst consists of 16 64-bit data words. The memory read base addresses of the four bursts are transmitted on the read channels in one single block. Subsequent transfers are already initiated without waiting for the preceding transfer to complete. This creates two continuous data streams and optimally utilizes the available external memory read bandwidth. The data streams are indicated by the red and blue arrows labeled with S1 and S2.

Connecting the units in an alternating fashion is an architecture detail that preserves a balanced load on the high-performance ports when the number of processing units is changed in the design.

AXI stream protocol implementation

The high-performance port interfaces follow the *Advanced eXtensible Interface*[11] (AXI) standard (Arm Limited, 2021); more precisely, they provide 64-bit AXI3 interfaces. The implementation of the AXI master interfaces of the DTM was tailored to efficiently read the LSTs from external memory and transfer the presynaptic data. The AXI protocol is based on so-called data *bursts* and *beats*, and it defines several channels for read and write signals. The implemented protocol uses two of these channels: the read address channel and the read data channel. Figure 3.12 illustrates this protocol showing the transmission of the presynaptic data for two spike events.

A complete LST is transmitted in two parallel sequences of four bursts, i.e., four bursts on each port, where a burst carries 16 x 64-bit data words, transmitted in 16 beats. The read address

[11] The Advanced eXtensible Interface (AXI) standard is an extension of the Advanced Microcontroller Bus Architecture (AMBA), which is an open standard.

channels describe the address and control information of the data bursts transferred on the read data channels. The implemented protocol aims to create streams of data that are as continuous as possible in order to make the best use of the available DDR memory bandwidth. The memory read base addresses of four data packet bursts are therefore transmitted in one single block and new transfers are initiated without waiting for the previous transfer to complete. Nonetheless, access latencies caused by arbitration and scheduling in the DDR memory controller can only be partially hidden. This is examined in more detail below.

External memory read efficiency

The theoretical maximum throughput (or bandwidth[12]) of the DTM is determined by the word length of the data streams S1 and S2, and the PL clock frequency. It can be calculated by

$$B_{\mathrm{DTM,max}} = 2 w_{\mathrm{S}} f_{\mathrm{clk}}, \tag{3.7}$$

where w_{S} denote the word length of a single stream and f_{clk} is the PL clock frequency. The maximum clock frequency at which the HNC node design can still meet the timing constraints is 200 MHz. At this clock frequency, and with $w_{\mathrm{S}} = 64$ bit, the maximum throughput results in

$$B_{\mathrm{DTM,max}} = 16\,\mathrm{B} \cdot 200 \cdot 10^6\ \mathrm{s}^{-1} = 3200\,\mathrm{MB/s}. \tag{3.8}$$

In practice, this is not achieved. A single spike event triggers the transfer of a 1KiB data packet. The transfer time measured here is approximately 550 ns[13]. This corresponds to 1862 MB/s and is the bandwidth utilized and *seen* by the RB FIFO buffers.

The DDR3[14] memories that the ZC706 development board carries can provide a theoretical peak memory bandwidth of $B_{\mathrm{DDR}} = 4264$ MB/s. The memory is organized as 4 x 256 Mbit x 8 bit (1 GiB) and clocked at 533 MHz. This results in a data rate of 1066 MT/s[14] (Micron, 2006) and the aforementioned peak memory bandwidth – 4 bytes can be transferred on each falling and rising edge of the clock signal.

The achieved external memory read efficiency is thus 43.7%, and the DTM can only utilize 58.2% of its possible bandwidth. Table 3.1 lists the calculated and measured bandwidths of the DTM for the three PL clock frequencies 100 MHz, 150 MHz, and 200 MHz, and shows the resulting bandwidth utilization efficiencies for the external memory and the DTM itself. At all

[12] The terms *bandwidth* and *throughput* are used interchangeably here.
[13] The time was measured using an external logic analyzer.
[14] Double Data Rate. DDR memories transfer data on both the rising and the falling edge of the clock signal. The measurement for the effective data rate is megatransfers per second (MT/s).

External DDR3	PL clock frequency	PS/PL Data Transfer Module		Efficiency	
B_{DDR}	f_{clk}	$B_{\mathrm{DTM,max}}$	$B_{\mathrm{DTM,meas}}$	DDR3	DTM
4264 MB/s	100 MHz	1600 MB/s	1280 MB/s	30.0%	80.0%
	150 MHz	2400 MB/s	1707 MB/s	40.0%	71.1%
	200 MHz(*)	3200 MB/s	1862 MB/s	43.7%	58.2%

(*) The maximum clock frequency at which the HNC node design can still meet timing constraints.

Table 3.1 | PS/PL Data Transfer Module and external DDR memory data transfer efficiencies. Listed are the calculated and measured bandwidths of the *PS/PL Data Transfer Module* (DTM) for three different PL clock frequencies, and the resulting efficiencies of the bandwidth utilization for: **DDR3**, the measured DTM bandwidth $B_{\mathrm{DTM,meas}}$ with respect to the theoretical DDR3 memory peak bandwidth B_{DDR}; and **DTM**, the measured DTM bandwidth $B_{\mathrm{DTM,meas}}$ with respect to the module's theoretical maximum bandwidth $B_{\mathrm{DTM,max}}$. At all three clock frequencies, the efficiency of the external memory stays below expectations and the DTM cannot utilize the theoretically possible bandwidth.

three clock frequencies, the efficiency of the external memory stays below expectations and the DTM cannot utilize the theoretically possible bandwidths.

These differences in measured and theoretical values can be attributed to latencies caused by arbitration and scheduling in the DDR memory controller of the XC7045 chip. These add approximately 200 ns to 250 ns to the transfer of a 1 KiB data packet. Nevertheless, the DTM achieves higher throughput than achievable using two (one per data stream) AMD Xilinx AXI DMA soft-IP cores (AMD Xilinx, 2019a) – the common solution here. The throughput of a single DMA soft-IP core is specified with 399.04 MB/s at 100 MHz clock frequency (AMD Xilinx, 2019a); hence two AXI DMA soft-IP cores clocked at 200 MHz would achieve approximately 1600 MB/s, i.e., 86.0% of the measured throughput of the DTM.

3.3.4.3 Presynaptic Data Processing – Ring Buffers

The ring buffers (RBs) are a key component of the HNC node. In the RBs, the synaptic inputs of neurons are accumulated (see Section 3.3.4.1 for the working principle). This input is derived from the presynaptic data that is processed in the RB pipelines. This processing is critical to performance and design decisions must be well thought out. Therefore, this section provides a more thorough architecture exploration. The functional relationships between RB, RB pipeline, and the components of a processing unit are described, and architecture alternatives are discussed.

In Figure 3.13A, the high-level architecture of a processing unit is shown, illustrating the interaction of its components. The RB pipeline fetches the presynaptic data items from the RB FIFO and accumulates them in the RB. The ODE solver pipeline retrieves these weighted synaptic inputs (i_{exc} and i_{inh}) from the RB and incorporates them into the computation of the

model dynamics, updating the neurons state vectors ($\boldsymbol{y}_k \leftarrow \boldsymbol{y}_{k-1}$) stored in the state variables buffer (SVB). The scheduling of the read and write operations performed here on the RB and SVB memories is relevant to data consistency as well as performance.

For example, both the RB and the ODE solver pipeline perform simultaneous read/write operations on RB and SVB, respectively (Figure 3.13B). In the case of the RB pipeline, reading an RB entry poses a risk of potentially missing previous RB update operations. In the following this is called a *read-before-write conflict*. The parallel execution of the two pipelines introduces concurrent RB read operations resulting in read contention (Figure 3.13C1). Also, the validity of an RB entry is an issue. The RB is written in a round-robin fashion and segments are reused, leaving RB entries in an invalid state.

These aspects need to be considered in the design. The functional requirements and technical characteristics of the selected FPGA-SoC technology here define the boundary conditions of the design space.

3.3.4.3.1 Architecture Exploration

For an architecture exploration, in the following, four aspects are considered and discussed: (i) simultaneous read/write operation; (ii) concurrent read access and read contention; (iii) potential read-before-write conflicts; and (iv) the validity of RB entries, i.e., synaptic inputs.

Simultaneous read and write operation

The processing units derive their ability to accelerate computations primarily from the pipelined operation and the locality of the data stored in fast on-chip BRAM memories. The RB pipeline and the ODE solver pipeline both perform simultaneous read/write operations on RB and SVB, respectively. This is illustrated in Figure 3.13B. Using single-port memory here will create a read-write contention problem. The sequential execution of reads and writes requires additional wait clock cycles to be included in pipeline operations. This increases a pipeline's *initiation interval*. The initiation interval is defined as the number of clock cycles elapsed between two consecutive iterations of a pipeline. It is of interest in a design because it determines the throughput of a pipeline. A pipeline with an initiation interval equal to 1 provides a maximum throughput and will deliver a result at its output every clock cycle.

To achieve this, we can take advantage of the true dual-port feature of the BRAM blocks. Read and write operations on BRAMs can be performed totally independent, sharing only the stored data. RB and SVB are therefore implemented using this type of memory allowing both the RB and the ODE solver pipeline to achieve an initiation interval equal to 1. However, similar throughput can also be achieved with single-port memory. This topic will be revisited in Chapter 6.

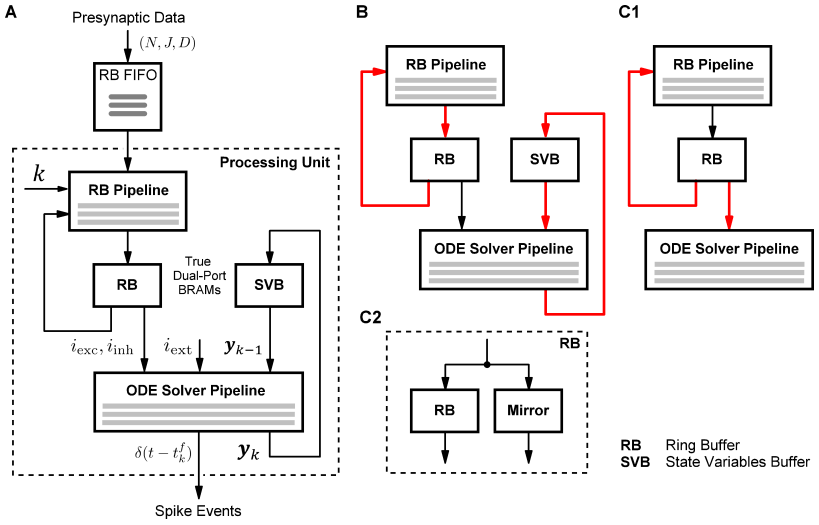


Figure 3.13 | Interaction of a processing unit's components. (**A**) High-level architecture of a processing unit and interaction of its components: The RB pipeline fetches the presynaptic data from the RB FIFO and accumulates the synaptic inputs in the RB. During a simulation time step, the ODE solver pipeline reads an entire RB segment to retrieve the weighted synaptic inputs i_{exc} and i_{inh} of all its assigned neurons and updates their states ($\boldsymbol{y}_k \leftarrow \boldsymbol{y}_{k-1}$) stored in the SVB. For optimal pipeline throughput, both RB and SVB use fast on-chip BRAM memories leveraging their true dual-port capability. (**B**) Shows the simultaneous read/write operations performed by the RB pipeline and the ODE solver pipeline on the RB and SVB memories (marked by read arrows). For the executed RB pipeline algorithm, these operations introduce the risk of potential read-before-write conflicts. (**C1**) Illustrates the problem of concurrent RB and ODE solver pipeline read operations (marked by read arrows), which introduces read contention on the RB. (**C2**) Shows a technical solution to the contention problem illustrated in (**B1**); here, a simple mirroring of the RB content. See the main text for explanation.

Concurrent read access and read contention

A concurrent execution of the RB pipeline and the ODE solver pipeline will cause read contention on the RB's read port because they both share this resource. Figure 3.13C1 illustrates this. A possible solution to avoid this contention is to use memory mirroring, where the RB pipeline maintains a copy of the RB content. The idea is illustrated in Figure 3.13C2. The second read port that is created in this way allows both pipelines to have independent read access. However, by implementing this architecture variant and comparing the achieved acceleration factors with and without serializing the RB pipeline and ODE solver pipeline RB read operations, it has been found that it is sufficient to execute operations sequentially. Without an asynchronous external spike input and high ODE pipeline iteration latency, which is not given (see Section 3.3.4.4), the

effect on performance is minimal. When presynaptic data items are placed into the RB FIFOs, only an early arriving spike event may find RB pipelines stalled due to RB contention. The additional latency is minimal, on the order of a few clock cycles per simulation time step, and thus negligible. Moreover, a minimal gain in performance is opposed here to a resource-intensive solution; mirroring the RB content doubles the size of the required BRAM blocks, which is a scarce resource. Note that the dual-port feature of the BRAM blocks remains valuable in the RB implementation, enabling efficient operation of the RB pipelines.

Potential read-before-write conflicts

There are two potential sources of RB read-before-write conflicts: (i) the ODE solver pipeline misses an RB pipeline update for an RB entry; and (ii) the RB pipeline itself misses an own RB update operation.

The ODE solver pipeline misses an RB update: In a simulation time step, the ODE solver pipeline of a processing unit processes an entire RB segment; the segment that is associated with the current time step k. The RB pipeline never addresses this segment when updating RB entries. The RB segment in which an entry is to be updated is derived from the synaptic delay value D as $k+D$, where D is a multiple of h and greater null, hence they are never the same. Nevertheless, the ODE solver pipeline may miss RB updates performed in the previous time step $k-1$. This has the following reason.

In order to keep the pipeline always filled, the first part of the segment processed in the k^{th} time step is already fetched into the pipeline at the end of the previous time step $k-1$ – the number of entries *prefetched* corresponds to the depth of the pipeline. *Late updates* in this part of the segment may therefore be missed by the ODE solver pipeline. This can only occur for synaptic events with $D = h$, i.e., RB updates targeting the subsequent time step $k+1$. Such an event invalidates the ODE solver pipeline, which then must be reset and restarted. This adds an additional latency to the processing, where the latency value depends on the depth of the pipeline and is thus model-specific.

Technically, the RB pipeline could determine whether a restart condition exists at the time a synaptic event is processed. There is another option, which requires minimal hardware support and a few software system enhancements. During connection setup, the *Connection Manager* can identify synaptic connections that might trigger a restart condition. This can be determined based on the synaptic delay value and the hardware resource assigned to the postsynaptic neuron, i.e., its position in the pipeline processing sequence. The HNC node has implemented this solution and encodes the restart trigger in the presynaptic data (see Appendix **D**). This information is extracted by the RB pipeline and passed to the main control FSM, which then initiates a restart of

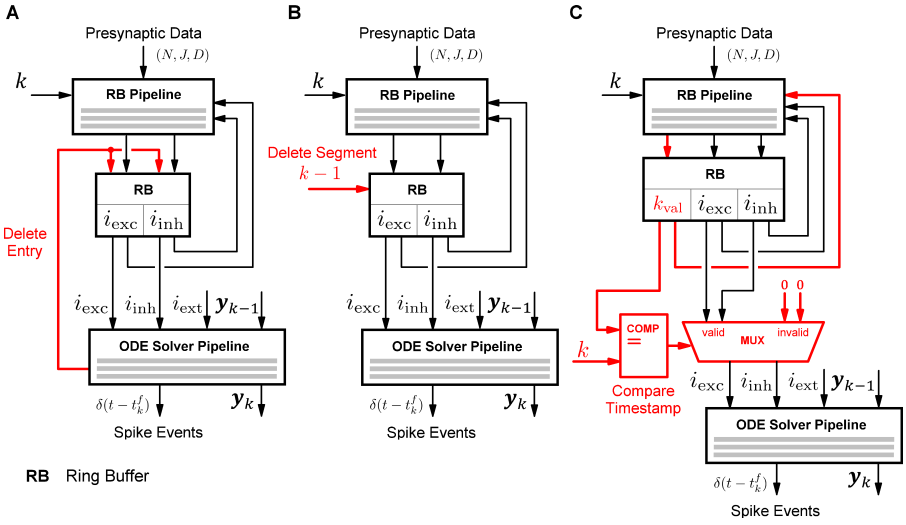


Figure 3.14 | Architecture alternatives to guarantee the validity of synaptic inputs. (**A**) The ODE solver pipeline deletes an RB segment entry immediately after it is processed. The values of i_{exc} and i_{inh} are set to zero and the entry can be reused. (**B**) In a simulation time step, the previous segment is deleted just before it is reused. (**C**) Each RB entry is associated with a timestamp that indicates the RB cycle for which the entry contains a valid synaptic input. The red outlines highlight the architectural differences. See the main text for detailed descriptions of the architecture alternatives.

the ODE solver pipeline.

The RB pipeline misses an RB update: It takes several clock cycles for a presynaptic data item to pass through the RB pipeline. While an RB entry for one data item is being written, the entry for the subsequent data item is already being read. If both the write and read operations access the same RB segment entry, the read operation will miss the previous update, resulting in an incorrect value. This can only occur if successive presynaptic data items target the same postsynaptic neuron and have the same synaptic delay. Therefore, only multapses[15] can run into this problem. By rearranging the corresponding elements in the LST during network instantiation, the issue can be solved in software.

Validity of RB entries and synaptic inputs

During a simulation time step, the ODE solver pipeline reads an entire RB segment to retrieve the synaptic inputs of all neurons assigned to the corresponding processing unit. After $k + K_{\text{RB}}$ time

[15] A synapse can be a *multapse*, which is a property of a synaptic connection that allows multiple synapses from a presynaptic to a postsynaptic neuron.

steps, i.e., a complete RB cycle, a segment is reused while the accumulated synaptic inputs from the previous cycle remain in the segment entries. These are now invalid. Figure 3.14 shows three architectures realizing different approaches to guarantee the validity of the RB entries. These approaches are: (i) delete an RB segment entry immediately after it is processed (Figure 3.14A); (ii) delete an entire RB segment before it is reused (Figure 3.14B); and (iii) mark an RB segment entry as valid for a specific time step when it is updated (Figure 3.14C). The technical advantages and disadvantages of the different solutions are discussed in the following.

Delete an RB segment entry immediately after it is processed: A resource-saving and probably the most obvious solution is to delete an RB segment entry immediately when it has been processed by the ODE solver pipeline. This is illustrated in Figure 3.14A. The downside of this solution is a higher processing latency. In each simulation time step, the ODE solver pipeline reads through an entire RB segment. This requires N^{RB} additional RB write operations, leading to contention on the RB write port for early arriving spike events. The deletion of the k^{th} segment, i.e., the segment associated with the current time step, also destroys the restart capability (see above). As a consequence, RB and ODE solver pipeline processing needs to be strictly serialized. The number of clock cycles that this serialization adds to the processing corresponds to the depth of the ODE solver pipeline.

Delete an entire RB segment before it is reused: There is no need to delete an RB entry immediately after it is processed. It is sufficient to delete a segment before it is reused. In time step k, this is the segment that was processed by the ODE solver pipeline in time step $k-1$. This solution has the advantage that the current segment remains intact, which preserves the restart capability. The disadvantage of this solution is again a higher latency. Sequential deletion of segment entries adds N^{RB} clock cycles to the processing. The segment to be deleted is also inaccessible to RB updates, which reduces the maximum possible synaptic transmission delay to $D_{\mathrm{max}} = h(K_{\mathrm{RB}} - 2)$.

Mark an RB segment entry as valid for a specific time step when it is updated: When read by the ODE solver pipeline, a segment is only valid for a specific time step. This is exploited by the architecture variant shown in Figure 3.14C. Each RB segment entry is associated with a timestamp that indicates the RB cycle for which an entry contains a valid synaptic input. This *valid timestamp* k_{val} is set when an entry is updated by the RB pipeline and verified when the entry is read by the ODE solver pipeline. The timestamp is derived from the calculated target simulation step as $k + D$ excluding the lower $\log_2(K_{\mathrm{RB}})$ digits. The timestamp is thus counting the RB cycles. In order to verify the validity of an entry before it is passed to the ODE solver pipeline, the value of k_{val} is compared with the corresponding bits of k. If they are equal, i_{exc} and

i_{inh} represent valid synaptic inputs. Otherwise, i_{exc} and i_{inh} are set to zero.

This solution preserves the ability to restart the ODE solver pipeline, and avoids additional RB write operations to clear segment entries. Comparing the three architecture alternatives, this solution achieves the best performance. The disadvantage of its implementation is that it entails a higher BRAM memory requirement to store the timestamps. The amount of this additional memory is determined by the timestamp's word length $w_{\text{len,kval}}$, which must be large enough so that no overflow can occur until all entries in an RB have received an update. If $w_{\text{len,kval}}$ is chosen such that the timestamp can cover the entire simulated time T_{sim}, an overflow becomes impossible. This word length can be calculated by

$$w^{\text{sim}}_{\text{len,kval}} = \lceil \log_2 \frac{T_{\text{sim}}}{hK_{\text{RB}}} \rceil. \tag{3.9}$$

As an example, for 20 minutes simulated time, a simulation resolution of $h = 0.1$ ms, and an RB depth of $K_{\text{RB}} = 256$, i.e., $D_{\text{max}} = 25.5$ ms, the resulting word length is $w^{\text{sim}}_{\text{len,kval}} = 16$ bit. In practice, this many bits are not needed.

To get a better sense of what word length is required and sufficient, we can also calculate the number of RB cycles that are expected to update all of the RB entries of a single RB at least once. If we further make the assumption that each incoming synaptic event updates an arbitrary RB entry and that entries are equally likely selected with the probability $1/K_{\text{RB}}$, we can ask for the expected number of events needed here. This question is equivalent to the question posed by the classic *Coupon Collector's Problem*[16]. The expected number of synaptic events needed to update each RB entry at least once can thus be calculated by

$$E(X) = K_{\text{RB}} \sum_{n=1}^{K_{\text{RB}}} \frac{1}{n}. \tag{3.10}$$

The frequency of updates determines the number of RB cycles and depends on the characteristics of the network model to be simulated, i.e., the firing statistics and number of synaptic connections. Using Equation (3.10), an estimate for the word length of the timestamp can be given as

$$w^{\text{est}}_{\text{len,kval}} = \lceil \log 2 (\frac{1}{\bar{\nu} \bar{K}_{\text{in}} h} \sum_{n=1}^{K_{\text{RB}}} \frac{1}{n}) \rceil, \tag{3.11}$$

where $\bar{\nu}$ denotes the average firing rate of the neurons in the network, and $\bar{K}_{\text{in}}$ is the neurons

[16] In probability theory, the *Coupon Collector's Problem* asks how many coupons you have to draw with replacements before each coupon is drawn at least once.

average in-degree. Referring to the example above and assuming an average firing rate of $\bar{\nu} = 10$ spks/s and a small network with an average in-degree of $\bar{K}_{\text{in}} = 100$, the word length of the timestamp results in $w^{\text{est}}_{\text{len,kval}} = 6$ bit. This is a far lower value than calculated for $w^{\text{sim}}_{\text{len,kval}}$ in the above example, and of course not sufficient. However, Equations (3.9) and (3.11) give us an estimation for a value range, based on which one can make a reasonable design decision. This range is defined by

$$w^{\text{est}}_{\text{len,kval}} < w_{\text{len,kval}} \leq w^{\text{sim}}_{\text{len,kval}}. \tag{3.12}$$

In practice, 10 bits proved to be sufficient for a small network with $\bar{K}_{\text{in}} = 100$. This has been verified by conducting a series of replicability tests. In these tests, a number of identical simulations were performed with the value of $w_{\text{len,kval}}$ systematically varied. The results were then compared for spike-for-spike identicality. The number of RB segments here was set to $K_{\text{RB}} = 128$, which corresponds to a maximum synaptic delay of $D_{\text{max}} = 12.7$ ms.
Note that a small network creates low workload (see also the workload model introduced in Section 5.2.1). For large networks much higher RB update frequencies are to be expected, which further reduces the required word length of the timestamp.

3.3.4.3.2 The Implemented Architecture Design

With respect to the design objective of achieving best possible performance, the ring buffers are a critical component. Accordingly, and with regard to value and resource requirements, the following design decisions were made:

- RB memories are implemented as true dual-port BRAMs. This allows simultaneous read and write operations for optimal RB pipeline throughput.
- The operation of the RB pipeline and ODE solver pipeline is serialized. The design decision here is to accept RB read contention as the impact on performance is negligible.
- In order to ensure the validity of synaptic inputs, but not to introduce any additional latencies, the architecture variant that uses a timestamp was selected. The word length of the timestamp, $w_{\text{len,kval}}$, was set to 10 bits.
- To achieve sufficient accuracy and deterministic results, synaptic inputs are stored using the s16.23 fixed-point data type.
- In the current prototype, it is not exploited that transmission delays of inhibitory synaptic connections are typically shorter. No distinction is made regarding the number of RB segments for excitatory and inhibitory synaptic inputs. Circular buffers are all the same size K_{RB}. This was more of a practical decision to avoid an unnecessarily complex design. It has no impact on performance.

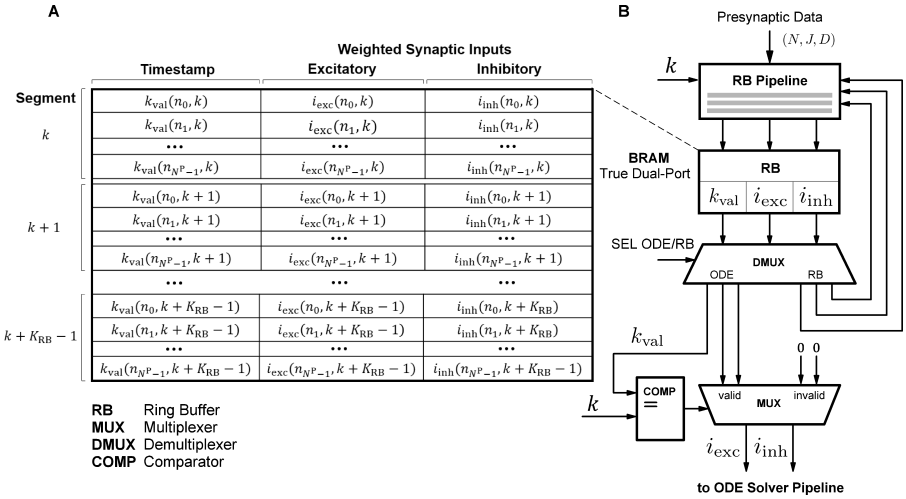


Figure 3.15 | RB architecture as implemented for the HNC node prototype. (**A**) RB memory layout. The RB memory is divided into K_{RB} segments. Each segment stores the weighted excitatory i_{exc} and inhibitory i_{inh} synaptic inputs of the N^{P} neurons assigned to a processing unit. Each pair of excitatory and inhibitory inputs is assigned a timestamp value k_{val} indicating its validity. (**B**) Architecture as implemented. RB and ODE solver pipeline read operations are serialized. The demultiplexer (DEMUX) selects the data path depending on whether the RB pipeline or the ODE solver pipeline requests read access to the RB. When the ODE solver pipeline reads an RB segment, its entries are tested for validity. For this purpose, the comparator (COMP) compares the current value of the time step k with the timestamp value k_{val}. If they are equal, the multiplexer (MUX) passes the synaptic inputs to the ODE solver pipeline, otherwise it sets them to zero.

The RB design as finally implemented is shown in Figure 3.15, where Figure 3.15A depicts the RB memory layout and Figure 3.15B details the microarchitecture. The RB pipeline works purely event-driven. If processing is not halted and the RB pipeline not stalled by concurrent ODE solver pipeline read operations, data items are immediately fetched from the RB FIFO and processed. The algorithm that the RB pipeline executes here is shown in the flow diagram in Figure 3.16.

RB pipeline throughput

In the implemented architecture, the RB pipeline has an initiation interval equal to 1. The throughput of an RB pipeline thus yields

$$B_{\mathrm{RB}} = f_{\mathrm{clk}} w_{\mathrm{len,LST}}, \tag{3.13}$$

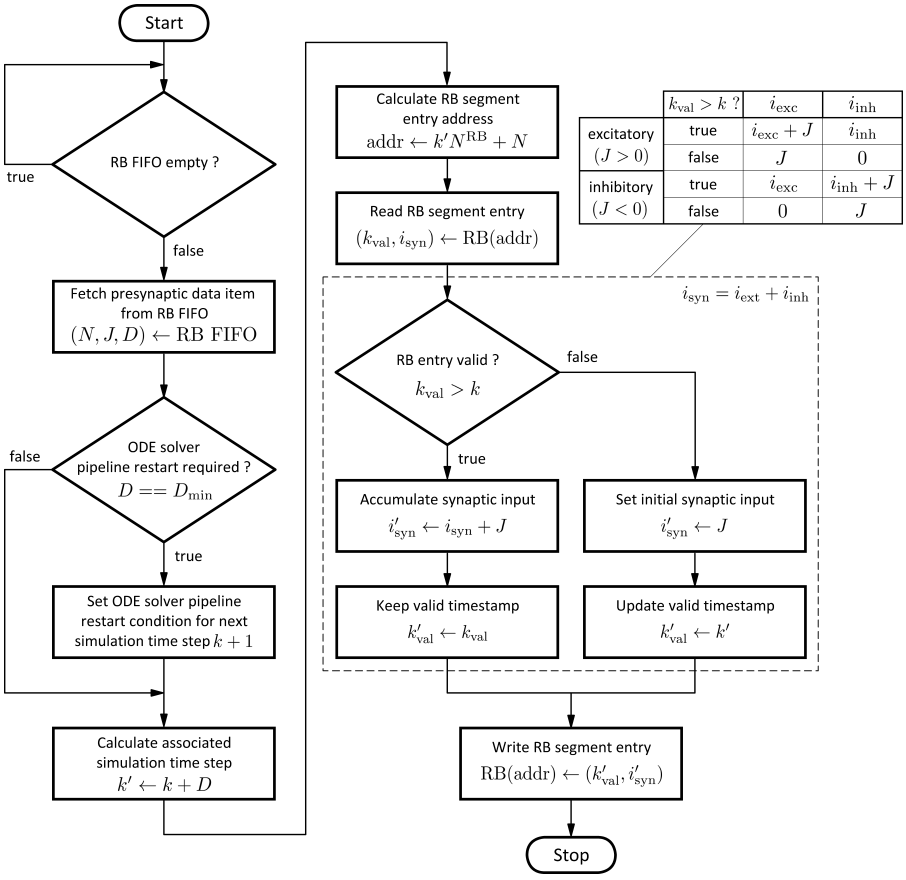


Figure 3.16 | RB update algorithm. Algorithm as implemented for the RB architecture shown in Figure 3.15. To simplify the illustration, the flow diagram shows the algorithm for unshaped synapses, where no distinction needs to be made between excitatory and inhibitory inputs – they can be lumped together. The full algorithm expands according to the table in the upper right, distinguishing between excitatory and inhibitory synaptic inputs.

where f_{clk} denotes the PL clock frequency, and $w_{\text{len,LST}}$ is the word length of an LST element, i.e., the data size of a presynaptic data item. At a PL clock frequency of $f_{\text{clk}} = 200$ MHz and with $w_{\text{len,LST}} = 64$ bit, this results in a throughput of $B_{\text{RB}} = 1600$ MB/s. Two processing units would therefore be sufficient to handle the bandwidth provided by the DTM, which was measured as $B_{\text{DTM,meas}} = 1862$ MB/s (see Section 3.3.4.2).

In order to keep the overall processing latency low, the HNC node parallelizes neuron state

updates using 16 processing units, each with its own RB and RB pipeline. Technically, all pipelines together would thus be capable of providing a total throughput of

$$B_{\mathrm{RB,total}} = PB_{\mathrm{RB}}, \tag{3.14}$$

where P is the number of processing units. This yields $B_{\mathrm{RB,total}} = 25.6\ \mathrm{GB/s}$, which corresponds to the processing of $32 \cdot 10^4$ RB update operations per 0.1 ms time step. This throughput would be sufficient to handle the workload produced by larger networks and still accelerate their simulation significantly. However, the HNC node prototype is here limited by the bandwidth of the external memory.

RB memory requirements

The RB memory accounts for most of the required BRAM resources. The size required for a processing unit's RB can be determined by

$$S_{\mathrm{RB}} = N^{\mathrm{P}} K_{\mathrm{RB}} (w_{\mathrm{len,kval}} + 2w_{\mathrm{len,s16.23}}), \tag{3.15}$$

where N^{P} denotes the number of neurons associated with a processing unit and $w_{\mathrm{len,s16.23}}$ is the word length of the s16.23 data type (40 bits). Accordingly, the total amount of RB memory required is given as

$$S_{\mathrm{RB,total}} = PS_{\mathrm{RB}}, \tag{3.16}$$

where P is the number of processing units. For example, the configuration $\{P = 16,\ N^{\mathrm{P}} = 64,\ w_{\mathrm{len,kval}} = 10\ \mathrm{bit},\ K_{\mathrm{RB}} = 128\}$ results in a total RB memory requirement of $S_{\mathrm{RB,total}} = 11.25$ Mbit. The XC7045 chip used in the prototypical implementation provides 19.2 Mbit of BRAM.

3.3.4.4 Neuron and Synapse Model Update – ODE Solver Pipelines

The core task of a simulation is to compute the model dynamics, i.e., to solve the ODE systems of many neurons. This task is performed by the ODE solver pipelines embedded in the processing units. In the following, the interrelationships of the components involved are described and the relevant architectural parts of a processing unit are detailed. In addition, an exemplary implementation of an ODE solver pipeline module is presented, specifically a module that implements the Izhikevich neuron model types.

3.3.4.4.1 Processing Unit Modes of Operation

The design of the HNC node is largely modular and component designs can be parametrized

using VHDL generics. For instance, the number of processing units is configurable. Within the processing units, ODE solver pipelines are implemented as RTL-modules and are exchangeable. A wide variety of neuron and synapse models can thus be supported. This flexibility in model implementation is also fostered by the use of generic data types and a model-independent interface. Figure 3.17A details the data input and output ports of an ODE solver pipeline module. Synaptic inputs use a 40-bit fixed-point data format, as explained earlier. State vectors, $\boldsymbol{y}$, are treated as generic 128-bit data words (larger word lengths are possible). Their interpretation is defined by a model-specific compound data type. This data type is determined by the model's properties, primarily the number of state variables and their required numerical precision – an approach that also allows for mixed precision.

Figure 3.17B illustrates how an ODE solver pipeline module is embedded in a processing unit. Two multiplexers (MUX) allow for the selection of different input and output data paths of the state vectors, thus providing four selectable operating modes for the processing units: (i) load state variables into SVB memories; (ii) unload state variables from SVB memories; (iii) run a simulation; and (iv) closed-loop operation of the ODE solver pipelines with the APU. The four operating modes are listed in the truth-table in Figure 3.17C. The truth-table shows how data paths are set depending on the multiplexers' select signals SEL1 and SEL2. The sequences of operations that the processing units perform in the different modes are steered by the main control FSM (not shown), which in turn is controlled by the HNC node software system. In the following, the four modes of operation are briefly described.

Load and unload state variables

For the data transfer between the APU and the processing units, a DMA controller is used; in particular, the AMD Xilinx DMA soft-IP core described in AMD Xilinx (2019a). The input and the output data paths of the state vectors are buffered in a series of interconnected registers. They form two 32 x 64-bit shift-registers (SHIFT REG) that chain the processing units together (Figure 3.17B). These registers are connected to the input and output ports of the DMA controller via two AXI streaming interfaces. When loading the state variables, the DMA controller copies the data of all neurons from external memory to the SVBs of all processing units, where the multiplexer settings are such that the ODE solver pipeline modules are bypassed. To unload the state vectors, the DMA controller transfers the data from the processing units' SVBs back to the external memory. The use of a DMA controller provides here an efficient way for copying the state variables to and from external memory. It also allows for a fast initialization of the BRAM memories in the FPGA. The entire DMA operation is controlled by the HNC node software system, as explained in Section 3.3.3.4.

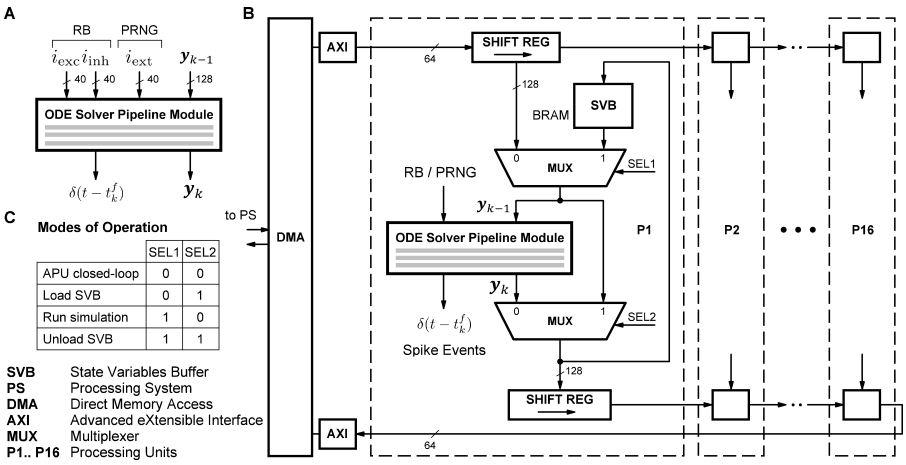


Figure 3.17 | Chained processing units and interaction of components. (**A**) ODE solver pipeline module data input and output ports. (**B**) ODE solver pipeline module as it is embedded in a processing unit. The processing units are chained together via two shift registers that are connected to a DMA controller for efficient data transfer with the PS. The state vectors $\boldsymbol{y}$ can be directed through different data paths depending on the configurations of the multiplexers (MUX) and the setting of the select signals (SEL1, SEL2). This allows four distinct modes of operation. (**C**) Truth table showing the combinations of SEL1 and SEL2 and the corresponding mode of operation.

Running a simulation

When running a simulation, data paths are set such that the SVBs are attached to the ODE solver pipelines. The SVBs are implemented as fast true dual-port BRAM memories; read and write operations of state vectors are performed concurrently, as explained earlier. In a single simulation time step, an ODE solver pipeline iterates over $N^{\text{P}} = 64$ neurons updating their state vectors ($\boldsymbol{y}_k \leftarrow \boldsymbol{y}_{k-1}$) stored in the SVB.

APU closed-loop operation

The fourth mode of operation allows the ODE solver pipelines to run in a closed-loop with the APU. For this purpose data paths are configured such that the shift-registers connect to the input and output ports of the ODE solver pipelines. The SVBs are not required in this setup. Instead, the DMA controller passes the state vectors directly through the ODE solver pipelines. In this mode of operation, the APU is included in the simulation, with the ODE solvers acting as hardware accelerators.

SVB memory requirements

Compared to the BRAM resource requirements of RBs, only a moderate number of BRAM blocks are required to implement the SVBs. The amount of memory is determined by the word length of the state vectors $w_{\text{len},\mathbf{y}}$ and the number of neurons processed on a node. It is given by

$$S_{\text{SVB}} = PN^{\text{P}}w_{\text{len},\mathbf{y}}, \tag{3.17}$$

where P denotes the number of processing units, and N^{P} is the number of neurons per processing unit. For the prototype, with $P = 16$, $N^{\text{P}} = 64$, and $w_{\text{len},\mathbf{y}} = 128$ bit, the required BRAM memory size is $S_{\text{SVB}} = 16$ KiB.

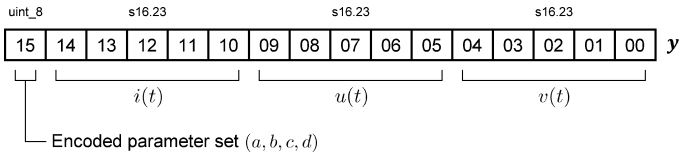
3.3.4.4.2 Example Implementation: Izhikevich Neuron Model

As an example of an ODE solver pipeline module, an implementation of the Izhikevich neuron model (Izhikevich, 2003) is presented in the following. A description of the model was given earlier in Section 2.3.1.1 (see also Appendix **A**).

The results of the rigorous verification and validation process presented in Chapter 2 have been used as the foundation for the design decisions made for the implementation. In particular, the choice of the data type used for state variables and the selection of a suitable integration scheme are motivated by the results of the calculation verification tasks conducted on the SpiNNaker system.

The arithmetic is implemented as 40-bit fixed point and uses the s16.23 data type for number representation. To solve the dynamics of the Izhikevich model ODEs, an explicit Forward Euler integration scheme with an integration step size of $h = 0.1$ ms is used. These design choices have been shown to meet the requirements in terms of numerical precision needed to achieve sufficient simulation accuracy (see Section 2.3.4.2).

For the implementation, we first need to define a suitable data format that maps the model's state variables to the generic 128-bit vector used in the ODE solver pipeline interface. Here, we define the following format:



The state variables $i(t)$, $u(t)$, and $v(t)$ are stored in three 40-bit fields in the s16.23 representation. The higher-order byte of the 128-bit vector is used to encode the selected set of Izhikevich

```
FOR EACH SIMULATION TIME STEP k:
  FOR EACH NEURON n:
    -- Input currents, stages: S1, S2 --
    I_n(t_k) := i_exc,n(t_k) + i_inh,n(t_k) + i_ext,n(t_k)
    i_n(t_k) := i_n(t_k-1)

    -- Forward Euler, stages: S2, S3, ..., S10 --
    v_n(t_k) := v_n(t_k-1) + h·[ 0.04v_n^2(t_k-1) + 5.0v_n(t_k-1) + 140.0 - u_n(t_k-1) + i_n(t_k-1) + I_n(t_k) ]
    u_n(t_k) := u_n(t_k-1) + h·[ abv_n(t_k-1) - au_n(t_k-1) ]

    -- Threshold detection, stages: S11, S12 --
    IF ( v_n(t_k) ≥ 30.0 ):
      v_n(t_k) := c
      u_n(t_k) := u_n(t_k) + d
      spikeEvent: δ(t - t_k^f)_n
    END
  END
END
```

Listing 3.1 | Algorithm (given as pseudo-code) corresponding to the hardware implementation of the ODE solver pipeline module shown in Figure 3.18. The algorithm implements the Izhikevich neuron model and uses a simple explicit Forward Euler integration scheme to solve the model dynamics.

model parameters, the neuron type. This data format is represented by a compound data type that consists of four components; an 8-bit unsigned integer and three s16.23 fields, which are themselves a custom data type. Based on this definition, the model can be made available to the HNC node software system by a corresponding extension of the *Neuron Manager*. This includes supporting routines for data type and endianness conversion. Data type definitions and supporting software components provide the basis for hardware design and implementation, and also the framework for verification to ensure that the ODE solver pipeline module interfaces correctly with the software system.

The design choices that have been made allow for an efficient implementation of the Izhikevich model. This is largely due to the use of fixed-point arithmetic, which does not introduce multi-cycle pipeline operations. The microarchitecture of the implemented module is shown in Figure 3.18. The design requires 6 multipliers (MULT), 8 adders (ADD), 2 subtractors (SUB), 1 comparator (COMP), 2 multiplexers (MUX), and 4 look-up tables (LUT). The corresponding algorithm is shown in Listing 3.1.

This implementation achieves a pipeline initiation interval equal to $\mathrm{II}_{\mathrm{ODE}} = 1$. The number of pipeline stages, or stage count, is SC = 12. Advancing the states for all neurons by one simulation

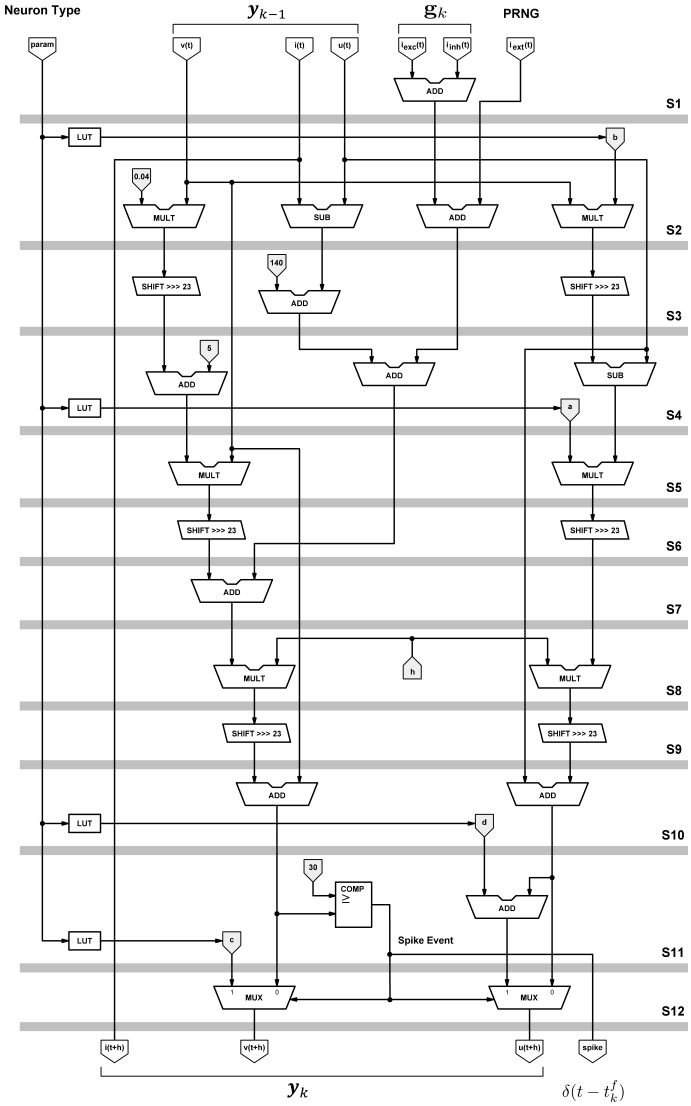


Figure 3.18 | Microarchitecture of the ODE solver pipeline module implementing the Izhikevich neuron model with static synapse. The gray bars indicate the pipeline registers. labeled S1 through S12 The pipeline achieves an initiation interval of $\mathrm{II}_{\mathrm{ODE}} = 1$. The corresponding algorithm is shown in Listing 3.1.

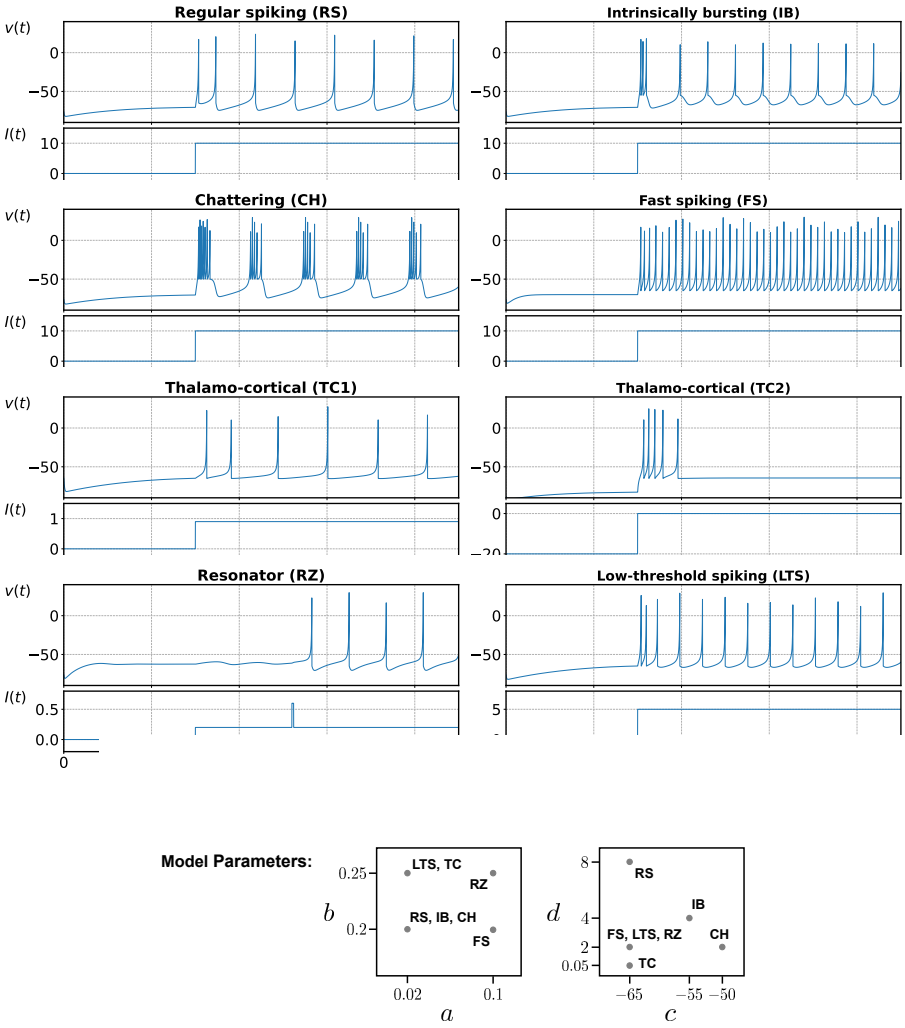


Figure 3.19 | Reproduction of the Izhikevich neuron model firing patterns. Shown are the different firing patterns of the Izhikevich neuron model reproduced on the HNC node with different choices of the model parameters (a, b, c, d) (parameters are shown at the bottom). The neurons were simulated for 450 ms while recording membrane potentials (shown for each neuron type in the upper panels) and stimulating the neurons with externally injected step currents and current pulses (shown for each neuron type in the lower panels). The results are in agreement with the firing patterns originally described in Izhikevich (2003).

time step requires

$$L_{\text{ODE}} = \text{SC} + \text{II}_{\text{ODE}} N^{\text{P}} \tag{3.18}$$

clock cycles. In Equation (3.18), N^{P} specifies the number of neurons associated with a processing unit. At a clock frequency of $f_{\text{clk}} = 200$ MHz and with $N^{\text{P}} = 64$, a complete state update of all neurons on a node takes only 380 ns.

The model implements a simple synapse with no shaping of the synaptic current $i(t)$ – the value is simply passed through the pipeline stages. The 8-bit parameter value (param) allows different sets of the Izhikevich model parameters (a, b, c, d) to be selected from the look-up tables. In this manner, one of the seven intrinsic firing patterns described in Izhikevich (2003) can be selected for each individual neuron. These patterns are: regular spiking (RS), intrinsically bursting (IB), chattering (CH), fast spiking (FS), thalamo-cortical (TC), resonator (RZ), and low-threshold spiking (LTS). The reproduction of these firing patterns on the HNC node is shown in Figure 3.19. To create the firing patterns, the neurons were stimulated with externally injected step currents and current pulses. These were generated by the software system, which intercepted and suspended the simulation, applied the external offset current, and then resumed the simulation.

A proof of correctness of this model implementation is given in Chapter 4.

3.3.4.5 Spike Events Processing – Serializer and Encoder

The HNC node uses an *Address Event Representation* (AER; Mahowald, 1992) to encode spike events. AER-based communication is well established in neuromorphic computing and the basis for efficient, low-latency spike-communication. In AER, a spike event is described by two values: the spiking neuron's location, i.e., its *address*, and the time the spike event occurred. In case of the HNC node, the location is defined by the node-id m_j and the local neuron-id $n_{\text{loc},j}$, the time is given by the simulation time step k in which the event occurred, or is represented by itself, respectively.

Due to the parallel running processing units up to $P = 16$ spike events can occur in a single clock cycle. In practice, such multiple events may happen only occasionally – more likely when higher firing rates are observed in the network. The spike events are stored as 16-bit vectors, here denoted as $\mathbf{b}^f$. In a single simulation time step, a processing unit processes N^{P} number of neurons; accordingly, up to N^{P} vectors may need to be processed here.

Figure 3.20A shows the high-level architecture with the components involved in this processing. The spike vectors $\mathbf{b}^f$ are first stored in a FIFO buffer (top left) along with the firing neurons position in the pipeline n_{pipe} and the current time step value k. As a data item is fetched from the FIFO buffer, the contained spike events are serialized while their corresponding processing unit

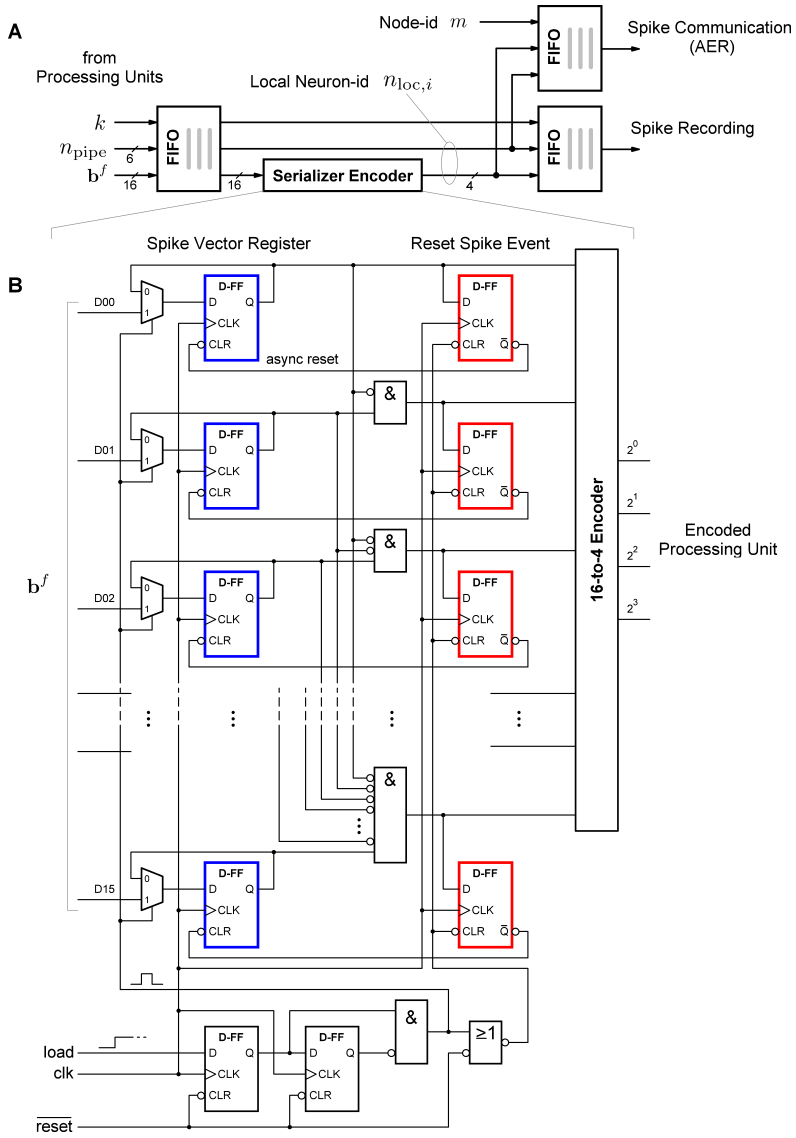


Figure 3.20 | Spike events processing. Spike events are serialized and encoded for further processing in recording and communication. Shown is: (**A**) the high-level architecture with the components involved; and (**B**) the schematic of the serializer and encoder circuit. See main text for description.

numbers are encoded as part of the local neuron-id of the spiking neurons: $(\mathbf{b}^f, n_{\text{pipe}}) \rightarrow n_{\text{loc},i}$. From the serialized events, for each event, two data items are created: one data item for use in spike recording; and one data item for use in communication. They are stored in separate FIFO buffers (top right). For communication, time is represented by itself. It is therefore sufficient to represent an event only by its location, i.e., the node-id m, and the neuron's local node-id $n_{\text{loc},i}$. For recording, the node-id can be omitted, but the time of the event is a mandatory part of the information; here this is stored as $(k, n_{\text{loc},i})$.

The latency of the spike processing is primarily determined by the serialization of events. The design has been optimized here, which resulted in a sequential circuit that works partially asynchronously. The circuit is shown in detail in Figure 3.20B. Its function is based on the use of two types of D-flip-flops (D-FFs): a D-FF type with asynchronous reset (marked blue); and a D-FF type with synchronous reset (marked red). The circuit operates as follows.

When a vector of spike events is fetched from the FIFO buffer, it is stored in the *Spike Vector Register* (marked blue), which is composed of 16 D-FFs with an asynchronous reset. Asserted outputs – the spike events – are selected by the logic gates bit by bit (FFs from top to bottom in Figure 3.20B) and passed to the 16-to-4 encoder circuit and the D-FFs on the right (marked red). The purpose of these D-FFs is to generate a signal that resets the corresponding bit in the spike vector. This repeats as many times as there are bits set in the spike vector. The asynchronous reset allows a bit in the spike vector to be cleared in the same clock cycle. The number of clock cycles required for the serialization and encoding process is then equal to the number of parallel spike events plus one clock cycle for loading the spike vector – the lowest possible latency achievable here.

3.3.4.6 Synaptic Delay Resolution – Address Counters

The HNC node always performs a simulation with the time resolution $h = 0.1$ ms. This value represents the temporal resolution of the simulation in the biological time domain, and is the *de facto* standard used in digital simulations of spiking neural networks in neuroscience. The HNC node updates neuron states and propagates spike events at this interval. The same time resolution is provided for synaptic transmission delays. However, not all network models require this resolution of synaptic delays. For example, the synaptic delay values in the two-population Izhikevich network model described in Section 2.3.1.1 have a resolution of 1 ms. When simulating this network, it would be sufficient to advance RB segments in steps of $10h$ instead of h.

The HNC node offers four different resolutions for synaptic delays, realized through different configurations of the RB segment address generation. The central components here are two

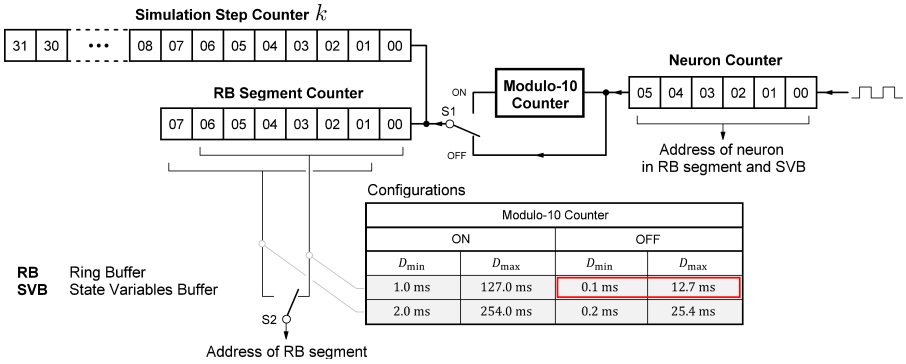


Figure 3.21 | RB and SVB address generation. Depending on the settings of the switches S1 and S2, the RB segment address, which is derived from the 8-bit *RB Segment Counter*, is advanced in steps of h, $2h$, $10h$, or $20h$ – a complete cycle of the 6-bit *Neuron Counter* corresponds to the interval h. This allows four different configurations of the minimum and maximum supported synaptic transmission delays $[D_{\text{min}}, D_{\text{max}}]$. The configurations and possible values are listed in the table, where the configuration marked red corresponds to the setting of the switches S1 and S2 as shown.

counters that generate the SVB and RB addresses and also advance the *Simulation Step Counter*. Figure 3.21 shows these components and illustrates their principle of operation.

In the interval h, the *Neuron Counter* performs a full cycle addressing all of the $N^{\text{P}} = 64$ neurons of a segment – this corresponds to a complete iteration of the ODE solver pipeline. Depending on whether the *Modulo-10 Counter* is on or off and which bits of the *RB Segment Counter* are selected, the RB segment address is advanced in steps of h, $2h$, $10h$, or $20h$. In this way, different values for D_{min} and D_{max} can be realized, where the resolution of synaptic delays is D_{min}. The table in Figure 3.21 lists the possible configurations. Note that the *Simulation Step Counter* is coupled to the *RB Segment Counter*. Therefore, when the *Modulo-10 Counter* is enabled, one simulation time step k corresponds to $10h$; ten iterations of the ODE solver pipelines. The ODE solvers then perform sub-steps where spike events are still propagated at the interval h. The coupling of the *Simulation Step Counter* to the *RB Segment Counter* also avoids rounding errors when calculating a synaptic event's RB target segment.

This feature allows the HNC node to be configured to accept larger synaptic delay values (up to D_{max}) without the need for an increase in the number of RB segments. This comes at the price of a reduced delay resolution, but can compensates to some extent for the confined RB size, which is limited by the number of available BRAM blocks.

3.3.4.7 Pseudo-Random Number Generation

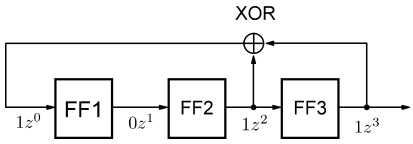
Random numbers are a necessary part of most neural network simulations and have a variety of uses. At network generation time, random numbers are often used to select and parameterize the connections to be created – it is common to describe network connectivity in the form of probabilities and probability functions. During simulation, random numbers are needed to generate stochastic network stimuli, e.g., in the form of Poisson spike trains or randomized input currents. Neuroscience simulations must be accurate and reproducible. Here, poorly generated random numbers can affect simulation outcomes. Therefore, high-quality random numbers that can be reproduced on demand are required.

Deterministic sequences of numbers that appear random are produced by pseudo-random number generators (PRNGs); deterministic algorithms of which a wide variety of types exist. PRNGs vary in complexity and the quality of the numbers generated. The characteristics that classify a PRNG as a high-quality number generator include, for instance: the uniformity of the generated numbers, i.e., numbers are equally probable; the ability to generate large sequences of numbers before a sequence is repeated; and that there is no correlation between subsequences. A classification of PRNGs and an empirical analysis of their quality was published, for example, in Bhattacharjee and Das (2022).

LFSR based PRNGs

For hardware implementation, the class of *linear feedback shift register* (LFSR) based PRNGs is particularly interesting. Introduced by Tausworthe (1965), its function is based on modulo-2 linear recurrence. LFSR PRNGs can produce long pseudo-random sequences and can be efficiently implemented in hardware. They achieve very high speed with very simple logic (see, e.g., Alfke, 1996; George and Alfke, 2007).

An LFSR PRNG can be implemented as a shift register consisting of m number of memory elements (flip-flops), where the outputs of some flip-flops – called *taps* – are XORed (or XNORed) and fed back as input to the register. Below an example of an LFSR with $m = 3$ memory elements and two taps is shown.



The configuration of the taps of an LFSR can be expressed as a polynomial modulo-2; the

coefficients are either 0 or 1. For the example above, the polynomial has the form $P(z) = 1 + z^2 + z^3$. The arrangement of the taps also determines the maximum sequence length. An m-bit *maximum-length* LFSR has a period of $2^m - 1$, which is the maximum length sequence of numbers it can produce.

PRNG implementation tailored for the HNC node

For the HNC node, a 111-bit maximum-length LFSR based PRNG was implemented, where the design was tailored to the simulation of the two-population Izhikevich network (see Section 2.3.1.1), i.e., to randomly select one neuron from the network at each simulation time step in order to supply it with an external input current. The architecture of the PRNG implementation is shown in Figure 3.22. For a maximum-length period, i.e., $2^{111} - 1$, the taps of the LFSR are 101 and 111 (according to George and Alfke (2007), Table 1). Instead of XOR, the LFSR uses XNOR in the feedback loop – the states of an XNOR-LFSR are the complement of the states of an XOR-LFSR resulting in an equivalent counter. The XNOR variant has the advantage that the LFSR will not *lock up* if the seed is zero.

The LFSR is used to address a 64K table, each entry of which is initialized with an integer value representing a local neuron-id $n_{\text{loc},i}$. The values were drawn from a uniform integer distribution such that $n_{\text{loc},i} \in \{1, 2 \ldots, 1000\}$. For this initialization, the Mersenne-Twister (mt19937) PRNG provided by the C++ Standard Library was used. The table serves two purposes: first, it translates the random LFSR bitstream into neuron-ids while limiting the range of values; and second, it adds additional randomness to the sequence.

The quality of this PRNG implementation has not been investigated further, e.g., by performing statistical tests such as those provided by the *Diehard test battery* (Marsaglia, 1995). Nevertheless, this implementation of a PRNG has proven to be sufficient for the task (see the substantiation assessment described in Section 4.5).

3.3.4.8 Synchronization

In a simulation, the temporal causality of spike events must be guaranteed. All spike events generated in a simulation time step k must have been delivered, and at least the RB updates must have completed for those synaptic events that have a delay value equal to the minimum temporal resolution, i.e., the events with $D = D_{\text{min}}$. This entails a synchronization of node-local processes before the next simulation time step $k + 1$ can be initiated. Apart from this, the nodes in a cluster are globally asynchronous. Their clock domains are not synchronized, and also individual update times may vary due to different load profiles (e.g., caused by differences in the

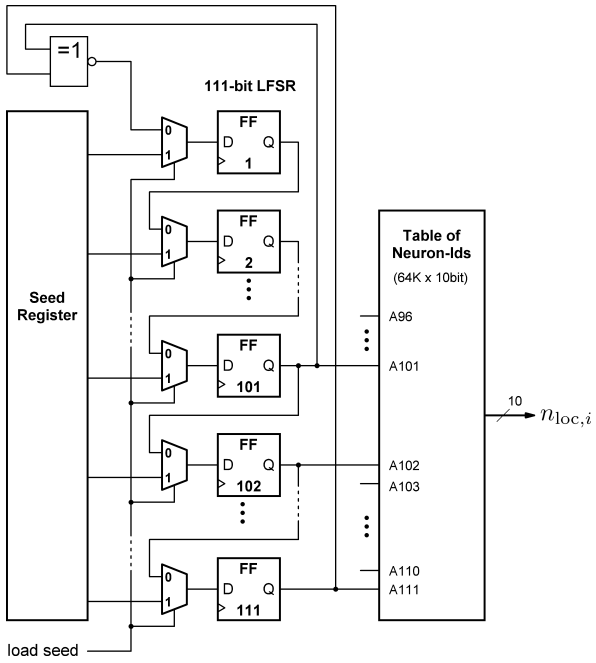


Figure 3.22 | Architecture of the implemented LFSR based PRNG. An 111-bit maximum-length LFSR (i.e., it has the period $2^{111} - 1$) with the characteristic polynomial $P(z) = 1 + z^{101} + z^{111}$ produces a random bitstream. The LFSR connects to a 64K x 10bit memory that acts as a look-up table, mapping a 16-bit value to a 10-bit local neuron-id $n_{\mathrm{loc},i}$. The look-up table was initialized using a Mersenne-Twister PRNG. The LFSR can also be initialized with a seed.

number of connections processed per spike event); hence, in a cluster, algorithmic time does not advance coherently. To achieve this, a synchronization at cluster level is also required.

Traditional approach

It is imperative that the above be considered in the design of a system that aims for accurate, reproducible, and even replicable simulations. The design choices made here are also relevant to performance. Trade-offs can be made if a certain degree of inaccuracy can be tolerated by the intended applications. The first generation SpiNNaker system (Furber et al., 2013), for example, does not have a global synchronization mechanism in place. The SpiNNaker update scheme rather *"enforces a finite minimum spike transit time"* (Furber and Bogdan, 2020, p. 110). This can lead to timeouts, especially at higher workloads, resulting in packets being dropped, und thus spike losses (see, e.g., van Albada et al., 2018). However, the field has evolved since, and

previous design decisions appear in a new light: *"In hindsight, this was a questionable* (design) *decision since computational neuroscientists can be more protective about their simulations!"* (Furber and Bogdan, 2020, p. 34).

A common solution to manage synchronization and guarantee the correct order of operations in multi-node systems is to use a *barrier* mechanism. Software simulators such as NEST (Gewaltig and Diesmann, 2007) use, for example, MPI[17] barrier message function calls for this purpose. The function blocks the calling process until all processes involved in the communication have also called it. The concept has also been adopted for neuromorphic hardware, where dedicated synchronization messages are sent over the same network over which spike events are propagated. For example, in Heittmann et al. (2022) the authors describe a message-based global synchronization scheme that is used in an implementation of the cortical microcircuit model (Potjans and Diesmann, 2014) on the IBM Neural Computer INC-3000 (Narayanan et al., 2020). The implementation uses a dedicated compute node (CN) that acts as a master node (MN) and central point to manage synchronization. The CNs in the system produce *barrier messages* that are sent to the MN. Once all CNs have reached the *barrier*, i.e., the MN has received a barrier message from all CNs, the MN broadcasts a *synchronization message* to release the CNs and allow them to proceed to the next simulation time step.

In Kauth et al. (2023), a synchronization scheme is proposed that the authors describe as *neighbor-to-neighbor* synchronization. It is applied to a similar implementation of the cortical microcircuit model, but uses a cluster of NetFPGA SUME boards. The described neighbor-to-neighbor scheme exploits the topology of the cluster, i.e., a two-hop worst-case latency. It also uses two events, but does not require a central synchronization point or master node.

The HNC node approach

Mixing barrier messages with spike communication puts a strain on the communication network. The HNC node design separates these two concerns, enabling solutions tailored to the task at hand. The solution proposed and implemented here retains the *barrier principle*, but uses dedicated hardware and *barrier signals* instead of a communication network over which barrier messages are sent.

There are two synchronization processes performed by the HNC node that need to be distinguished: (i) an intra-node synchronization process; and (ii) an inter-node synchronization process. For the latter, only a description of the conceptual idea is provided here, as only a single HNC node prototype currently exists. Technically, however, the necessary functionality to synchronize processes has been implemented for the HNC node, including synchronization at the cluster level.

[17] Message Passing Interface, https://www.mpi-forum.org/

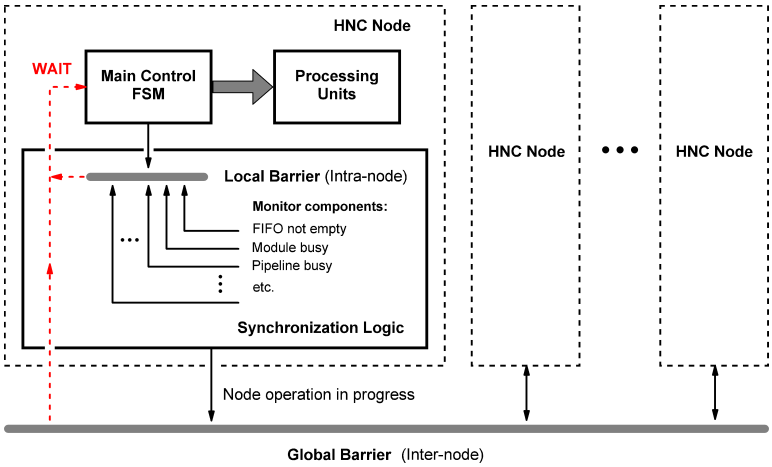


Figure 3.23 | Intra- and inter-node synchronization. An HNC node performs two synchronization processes at the end of a simulation time step: a node-local (intra-node) synchronization process; and a synchronization process at the cluster level (inter-node). For this purpose, a combinational circuit (box labeled *Synchronization Logic*) monitors the operation status of components and maintains two barrier signals; indicated by the gray bold lines labeled *Local Barrier* and *Global Barrier*. If asserted, the *Main Control FSM* is placed in a wait state, which prevents a HNC node to advance to the next simulation time step; illustrated by the red arrows. Only when all nodes in the cluster have completed operation, barriers are released and simulation time advances.

Figure 3.23 illustrates the working principle.

Intra-node synchronization: A combinational circuit continuously monitors the operating status of the components of an HNC node (in Figure 3.23, marked by the box labeled *Synchronization Logic*). It creates a node-internal *local barrier* signal (represented by the bold line labeled *Local Barrier*). If it is asserted, the HNC node's *Main Control FSM* will be placed in a wait state at the end of a simulation time step, preventing the node from proceeding to the next simulation time step (illustrated by the red dashed arrows). This barrier is released when all pipelines and modules have completed their operation, all spike events have been delivered, no RB updates are pending, and all FIFO buffers are empty. In this way, hardware blocks and processes synchronize within a node.

Inter-node synchronization: In a cluster of nodes, a similar mechanism is established by a *global barrier* signal (in Figure 3.23, represented by the bold line labeled *Global Barrier*) that ensures that simulation time advances coherently for all nodes. This barrier is released when all nodes in the cluster have completed intra-node synchronization. For this purpose the *Synchronization*

Logic also generates a signal that indicates when node operation is in progress and not completed.

Compared to message passing, this approach provides a very lean solution that also promises low latency. Technically, the global barrier signal could be realized by a simple wired-OR; in a larger cluster of nodes, by a hierarchical network to limit fan-in and fan-out.

3.3.5 Operating Latencies

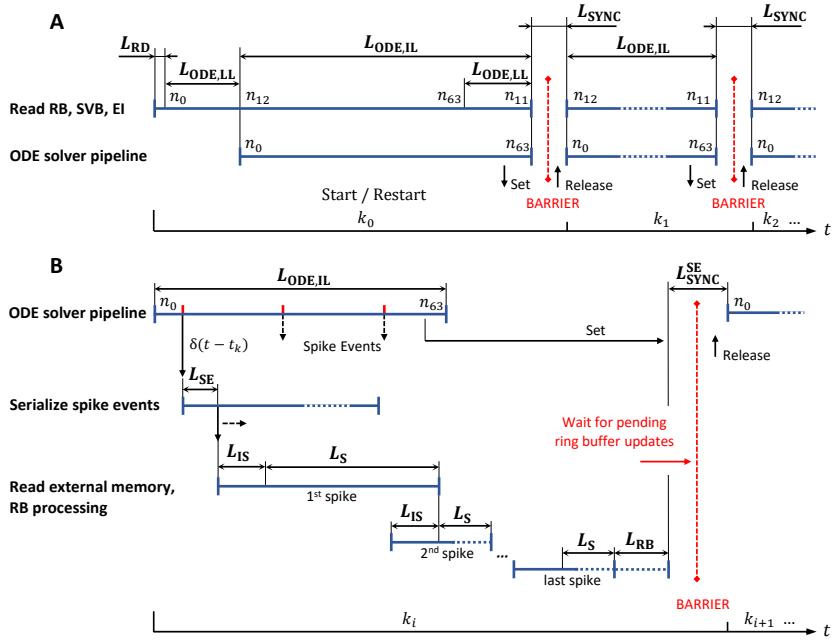
An essential part of the design work was the optimization of the microarchitecture to minimize latencies for best possible performance. In order to further investigate the design on system level in this respect, the operating latencies are extracted from the microarchitecture in the following. This forms the basis for the formulation of an accurate performance model, which will allow to systematically evaluate the performance characteristics of the HNC node for arbitrary workload situations, but also to predict the behavior of a cluster system (see the performance assessment presented in Chapter 5).

There are basically two different sequences of operations performed by the HNC node in a simulation time step that need to be considered for simulation performance: the sequence of operations performed when no spike event occurs; and the sequence of operations performed when spike events occur. The first consists mainly of the operation of the ODE solver pipelines and the node-local synchronization process at the end of a simulation time step. The sequence is shown in the timing diagram in Figure 3.24A. This sequence expands as spike events occur and get processed. The serialization of node-local spike events, the reading of the presynaptic data from external memory, and the processing of the spike events in the RB pipelines introduce additional latencies here. The sequence is shown in Figure 3.24B. The table in Figure 3.24C gives a summary of the latencies, their values in number of clock cycles, and provides a brief description of their meanings.

At simulation start (and also at ODE solver pipeline restart), the ramp-up of the ODE solver pipelines introduces two latencies: L_{RD}, which is a single clock cycle to initially read memory; and $L_{\mathrm{ODE,LL}}$, which is the latency caused by filling the ODE solver pipelines. The value of $L_{\mathrm{ODE,LL}}$ depends on the number of pipeline stages, and is thus a model-dependent latency. In a simulation time step, an ODE solver pipeline iterates over the N^{P} neurons associated with a processing unit. This results in the pipeline iteration latency $L_{\mathrm{ODE,IL}}$, which is given by

$$L_{\mathrm{ODE,IL}} = \mathrm{II}_{\mathrm{ODE}} N^{\mathrm{P}}, \tag{3.19}$$

where $\mathrm{II}_{\mathrm{ODE}}$ denotes the initiation interval of the ODE solver pipeline. At the end of a simulation



C

		Clock Cycles	Description
L_{RD}		1	Initial read latency; reading ring buffers (RB), state variables buffers (SVB), and external input buffers (EI)
$L_{ODE,LL}$		12	Initial load latency of the ODE solver pipeline; model specific; Izhikevich model implementation: stage count $SC = 12$
$L_{ODE,IL}$		64	Latency of ODE solver pipeline iterating over all neurons associated with the processing unit, N^P; equal to $II_{ODE} N^P$; $N^P = 64$; II_{ODE} pipeline initiation interval; model specific; Izhikevich model implementation: $II_{ODE} = 1$
L_{SYNC}		3	Synchronization latency when no spike event is processed
L_{SE}	L_{SEP}	4	Parallel spike events FIFO buffer latency
	L_{SES}	2	Latency serializing and packing spike events
	L_{SEF}	2	Latency of the spike events FIFO buffer
L_{IS}	L_{ISCAL}	4	Latency calculating memory read addresses and initiating PS to PL data transfers
	L_{ISADR}	9	Latency of the read address channel
L_S		≈80 / ≈90 / ≈110	Per spike event data stream latency; $C_{max}^M = 128$ local target connections; measured at f_{clk}= 100 / 150 / 200Mhz
L_{RB}	L_{RBF}	2	Latency of RB FIFO buffer
	L_{RBP}	5	Latency of RB pipeline
L_{SYNC}^{SE}		8	Synchronization latency when spiking events are processed

Caption overleaf.

Figure 3.24 | Operating latencies. The timing diagrams show the two basic sequences of operations: (**A**) simulation time steps in which no spike event occurs; and (**B**) simulation time steps in which spike events occur. The latter introduces additional latencies caused by the processing of spike events. (**C**) Overview and brief description of operating latencies, listing the number of required clock cycles. See also the main text for description.

time step, a number of clock cycles are required for synchronization, which is described by the latency L_{SYNC}.

Spike events can occur on any clock cycle during ODE solver pipeline operation. They are serialized, where two events are packed together. The latencies introduced by this process are summarized in L_{SE}. The table in Figure 3.24C breaks this down further. Each spike event initiates a sequence of operations in which the presynaptic data is retrieved from external memory and passed through the RB pipelines. This introduces the latencies L_{IS}, L_{S}, and L_{RB}. Here, the value of L_{S} cannot be derived from the microarchitecture. The latency L_{S} describes the *per spike data stream latency*, which is the average number of clock cycles required to read the presynaptic data of a single spike event, i.e., to retrieve an LST from external memory. We can derive this latency from the measured bandwidth of the *PS/PL Data Transfer Module* (DTM) (see Section 3.3.4.2) as

$$L_{\mathrm{S}} = C^{\mathrm{M}}_{\mathrm{max}} w_{\mathrm{len,LST}} \frac{f_{\mathrm{clk}}}{B_{\mathrm{DTM,meas}}}, \tag{3.20}$$

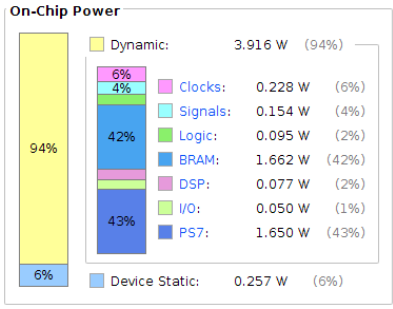
where $B_{\mathrm{DTM,meas}}$ is the measured bandwidth of the DTM at the PL clock frequency f_{clk}, $C^{\mathrm{M}}_{\mathrm{max}}$ denotes the maximum number of local target connections of a presynaptic neuron, and $w_{\mathrm{len,LST}}$ is the word length of an LST element, that is, the data size of a presynaptic data item (see Section 3.3.3.2).

At the end of a simulation time step in which spike events have been processed, there may still be data items remaining in the RB pipelines. These pending RB updates must be completed, which requires additional time for the node-local synchronization and introduces the latency $L^{\mathrm{SE}}_{\mathrm{SYNC}}$. In a multi-node system, the additional time required for inter-node synchronization would also add to the total latency. This is not explicitly included in the timing diagrams, but it is indicated in Figure 3.24 by the barriers marked by red dashed lines.

3.3.6 Breakdown of FPGA Resources and Power Consumption

On the left side of Figure 3.25 a breakdown of the FPGA logic resources utilized by the HNC node prototype implementation is shown. Immediately noticeable is that the implementation uses more than 80% of the BRAM blocks. This is primarily due to the memory requirements of the RBs, which have been sized to maximize the supported synaptic delay. The 368 DSP blocks are

Resource	Utilization	Available	Utilization %
LUT	39174	218660	17.92
LUTRAM	9363	70400	13.30
FF	49092	437200	11.23
BRAM	454	545	83.30
DSP	368	900	40.89



Total On-Chip Power: 4.186 W

Figure 3.25 | Resources utilization and power report. Utilization of the programmable logic part of the XCZ7045 AMD Xilinx Zynq SoC device for the implemented HNC node prototype (left), and the AMD Xilinx Vivado on-chip power report for a PL clock frequency of $f_{\text{clk}} = 200$ MHz (right).

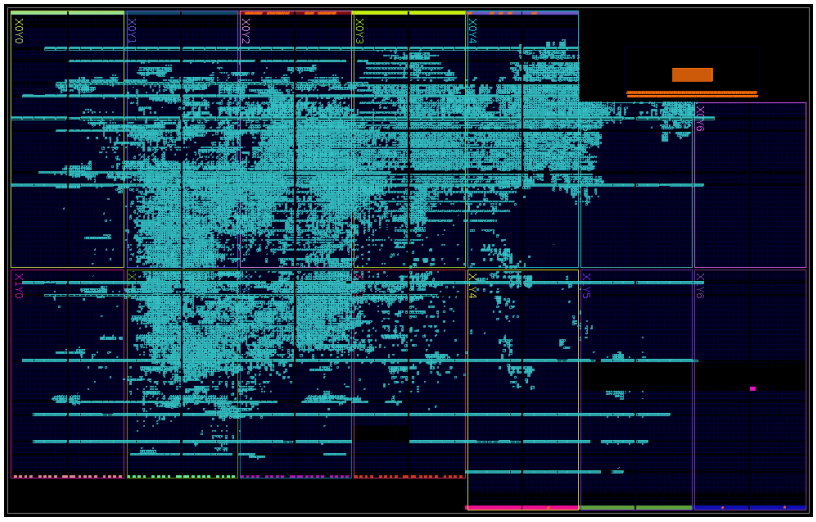


Figure 3.26 | Chip layout of the HNC node implementation on the XC7045.

solely used by the Izhikevich neuron model implementation of the ODE solver pipelines (see Section 3.3.4.4.2). Nevertheless, the design uses only about 14% of the available LUT, LUTRAM and FF resources. The chip layout with the area footprint is shown in Figure 3.26.

On the right-hand side of Figure 3.25, the power analysis report is displayed as generated by the AMD Xilinx Vivado design tools. It shows that 86% of the energy consumption is accounted for by on-chip memories and the Processing System, and this in roughly equal parts. For the total

on-chip power consumption the Vivado tools estimated 4.166 W. Note that this estimate is based on the XC7045 chip resources utilized and has a lower confidence level than an actual power measurement. The power report also does not include the power consumption of the external memory.

3.4 Discussion

Making neuromorphic computing a useful tool for neuroscientists is a demanding technical challenge, not only because of the conflicting requirements of efficiency and flexibility, but also due to the need for numerical accuracy and reproducibility. The prototyping of the HNC node revealed where the technical challenges lie and which aspects demand special attention from both an architectural and a technological point of view. Regarding the latter, the development was also an exploration of commercial off-the-shelf FPGA-SoC technology. It has been found that memory architecture, closely linked to semiconductor technology, plays a crucial role here, imposing design constraints that largely determine performance and possible system size, i.e., it affects the scalability of a system (for these aspects, see also Chapter 5 and Chapter 6).

Nevertheless, the development also demonstrates the great potential that FPGA-SoC technology holds as a substrate for neuromorphic computing. Design space exploration, the elaboration and evaluation of architecture variants, and the optimization of designs were central development tasks. The preceding sections reflected on these efforts, detailing an architecture of a neuromorphic compute node that can meet the demanding requirements of neuroscience simulation.

Development environment and logic design methodology

The HNC node design integrates hardware and software components whose functions are interdependent. This increases the complexity of the development process. The setup using the ZC706 development board here proved to be an effective and practical development platform, providing the tools necessary for efficient hardware-software co-development and co-verification. The chosen logic design methodology – a combination of RTL-level design and graphical block design – allowed the Zynq-7000 SoC architecture to be fully exploited and designs to be optimized at the lowest architectural level. The implementation in VHDL at the RTL level is rather error-prone. Therefore, hardware-software co-development has always been accompanied by a co-verification process. A comprehensive description of the verification approach is given in Chapter 4.

Flexibility

Flexibility and efficiency are opposing goals. Both are key requirements for a neuromorphic

platform dedicated to neuroscience simulation. The hybrid concept of the HNC node offers a good compromise here, taking advantage of the Zynq-7000 SoC device technology. The tight coupling of a general-purpose processor with a programmable logic device in a single chip allows an application to combine the flexibility of a software-based solution with the efficiency of application-specific hardware. In this regard, the HNC node can provide a level of flexibility that allows the system to adapt to changing requirements and cope with the rapid developments in neuroscience.

The need for flexibility arises mainly from modeling. A plethora of neuron and synapse models exist and new models are being formulated. Domain-specific languages (DSLs) such as NeuroML (Gleeson et al., 2010), NMODL (Hines and Carnevale, 2000), and NESTML (Plotnikov et al., 2016) have been developed to conveniently describe the dynamics of neuron and synapse models, where tools then generate simulation codes from a DSL description. The use of generic data types and the modularity in the hardware and software architecture design of the HNC node takes into account the requirements of these tools and workflows. This provides the technical prerequisites to make the system amenable to existing code generation techniques (see, e.g., Blundell et al., 2018a); here the generation of hardware descriptions for the ODE solver pipelines, and corresponding codes for *Neuron Manager* functions. The use of a High-Level Synthesis (HLS) description as code generation target appears to be an attractive option here. Not only can this be seamlessly integrated with existing tools, but such an approach could also take advantage of HLS' ability to find non-obvious pipelining scheduling schemes. By this means a wide variety of neuron and synapse models can be supported.

The HNC node architecture is open to extensions. Such an extension can be, for example, plasticity algorithms. Although plasticity models were deliberately omitted from the current HNC node prototype, they were considered when making design decisions. To enable the implementation of spike-based plasticity rules (Morrison et al., 2008), synaptic weights as well as spike events are stored in external memory accessible by both the APU and the programmable logic. The conceptual idea behind this is to use the APU's processor cores as dedicated *plasticity processors* that can also be combined with supporting accelerator hardware blocks. Plasticity rules and algorithms require a high degree of flexibility in algorithmic implementation, as this is a rapidly evolving area of research. Using general-purpose processors for this task is therefore a reasonable choice. The HNC node here can provide a flexible platform for fast prototyping and the exploration of novel architecture designs and algorithms.

Cluster operation

A goal of the HNC node development was to provide a scalable architecture, including the ability

to cluster multiple HNC nodes to form a neuromorphic computing system. Since only a single node prototype exists at this time, some concepts are not fully developed, such as an architecture for an efficient inter-node communication network. Nevertheless, cluster operation has guided design decisions and conceptual ideas, especially those related to spike communication and synchronization. Both are relevant to performance.

The HNC node design considers three types of communication with different latency and bandwidth requirements. These communication types are: inter-node spike communication, inter-node synchronization, and external communication. Inter-node spike communication and synchronization require ultra-low latency communication, but not high bandwidth. For external communication, high bandwidth is a desirable feature to minimize system setup time, e.g., when loading a network's connectivity data. Technically, modern FPGA-SoC devices offer a variety of standard peripherals that enable efficient chip-to-chip communication. These peripherals include high-speed serial gigabit transceivers (GTX/GTH), PCI Express, and low-voltage differential signaling (LVDS) user I/Os.

Conceptually, the HNC node separates the different communication concerns by targeting three different solutions tailored to each communication task.

Inter-node spike communication: A number of concepts and solutions exist for low latency inter-node spike communication, with the use of an AER communication protocol being the standard approach. This protocol is also used by the HNC node.

A concept for a balanced multi-hop communication architecture, which considers the requirements of large-scale neuroscience simulations, is presented in Kauth et al. (2020). In Moore et al. (2012) a 64-node FPGA cluster is described that uses high-speed serial links. The communication network of the cluster, which is organized in a 3D torus topology, achieves a hop-latency of 50 ns. A 35-node FPGA cluster with a network topology that has a two-hop worst-case latency is described in Kauth et al. (2023).

Inter-node synchronization: The HNC node provides a one-wire barrier signal solution for inter-node synchronization that does not require barrier messages to be exchanged between nodes. This is, to the best of my knowledge, a novel approach.

External communication: For external communication, the HNC node uses the 10/100/1000 Mb/s tri-speed Ethernet PHY provided by the Zynq-7000 SoC device and the TCP protocol. In the current prototype, this interface is only used to stream the recorded simulation data to a host system.

System integration

The architecture design of the HNC node also aims for seamless system integration, facilitated by modularity and the provision of software interfaces. Tools such as PyNN (Davison et al., 2009) and PyNEST (Eppler et al., 2009), which allow convenient network description, can be easily integrated. This has already been used during development to conduct joint hardware-software co-verification tasks (see Section 4.3). To prepare these tasks, the connectivity data of a test network was generated using PyNEST. This data was then imported into the HNC node where *Neuron Manager* and *Connection Manager* instantiate the network and allocate the necessary hardware resources.

The hybrid design provides the flexibility needed to integrate a neuromorphic system built from HNC nodes into complex simulation pre- and post-processing workflows. A thorough discussion on system integration, including the presentation of a concept for the integration into the high-performance computing (HPC) landscape, is given in Section 6.4.

The development has shown that commercial off-the-shelf FPGA-SoC technology has great potential as a substrate for neuromorphic computing. The reconfigurable logic allows extensive freedom in the implementation of numerical models, while the general-purpose processor cores provide an elegant way to execute performance non-critical tasks and allow for seamless system integration.

Chapter 4

HNC Node Implementation Correctness

4.1 Introduction

The HNC node has a hybrid hardware and software mixed architecture. Hardware and software components form a joint system whose design and implementation require hardware-software co-development. The same applies to the verification of hardware and software components as well as the validation at system level. The co-development process needs to be accompanied by a co-verification process. Tasks are very different here. Developing and testing a software component is fairly different from, for example, maintaining a finite-state machine model throughout hardware development.

The HNC node software system is written in C and almost all hardware components are developed in VHDL. In contrast to a High-Level Synthesis (HLS) approach, where a hardware design is formulated at the algorithmic level, e.g. in C, and the synthesis toolchain generates a hardware description in a reliable process, the implementation in VHDL at the RTL level is rather error-prone. A well thought-out test strategy is therefore essential. It must consider the verification of the correctness of the technical implementation of the hardware and software components, their correct interplay, as well as the validation of the entire system function; here it is the correctness of the outcome of the simulations performed on the HNC node. These activities must accompany the development work and be carried out as part of the co-development process.

In this chapter, the verification and validation methods described in Chapter 2 are revisited and applied to the HNC node to provide a proof of correctness. First, some calculation verification tasks are presented that were conducted to ensure the appropriateness of the design decisions made regarding the implementation of numerical algorithms. Second, the strategy that has been used for the functional verification of hardware components is described. Here I present a rather unusual approach that will be referred to as *in-FPGA verification*. This approach takes advantage of the FPGA-SoC architecture to construct a fully software-driven testbench. Finally, a substantiation assessment is performed. For this proof of correctness, the minimal two-population Izhikevich network, which has already been used in Chapter 2 to demonstrate a rigorous verification and validation workflow, serves as a test-case model.

Contributions

- The verification and validation methods presented in Chapter 2 and demonstrated by conducting a rigorous model verification and validation process are applied to the development of the HNC node, a novel neuromorphic architecture, to ensure and demonstrate the correctness of the implementation.

- An approach to hardware verification is presented that leverages the FPGA-SoC device architecture to implement a software-driven testbench. This approach is referred to here as *in-FPGA verification*. Methods and practices well established in software development, such as continuous testing and test-driven development, can thus be applied to the hardware-software co-development and co-verification processes without much effort.

4.2 Calculation Verification

The arithmetic operations for processing the synaptic inputs and for computing the neuron model dynamics are realized in programmable logic and work with custom data types, where the arithmetic is implemented as fixed point. The corresponding components, i.e., RB pipelines (Section 3.3.4.3) and ODE solver pipelines (Section 3.3.4.4), therefore require special attention with regard to their level of numerical error. Value range and numerical precision of a data type, algorithms, and the technical implementation are critical and a potential source of errors.

4.2.1 Value Range and Numerical Precision

The calculation verification task conducted in Section 2.3.4.2 revealed that a 32-bit signed fixed-point data type in an $s16.15$ representation does not provide the necessary numerical precision to capture the dynamics of the Izhikevich neuron model (Izhikevich, 2003) with sufficient accuracy. While this can be overcome by a higher precision $s8.23$ representation, it comes at the cost of a very limited value range. This motivated the use of the 40-bit data type in the $s16.23$ representation. The data type combines the value range of $s16.15$ with the precision of $s8.23$. The value range of the $s16.23$ data type is

$$-2^{16} = -65536 \quad \text{to} \quad 2^{16} - 2^{-23} = 65535.999999880. \tag{4.1}$$

This is sufficient to cover the value ranges of synaptic currents, state variables, and intermediate results of arithmetic operations when calculating the model dynamics of the Izhikevich neuron model. By providing eight additional fractional bits – compared to the $s16.15$ data type – calculations can be performed with sufficient accuracy (see Section 2.3.4.2).

4.2.2 Numeric Integration Scheme

Also the choice of the integration scheme is motivated by the results of the calculation verification task described in Section 2.3.4.2. An explicit numerical integration scheme is adequate for the

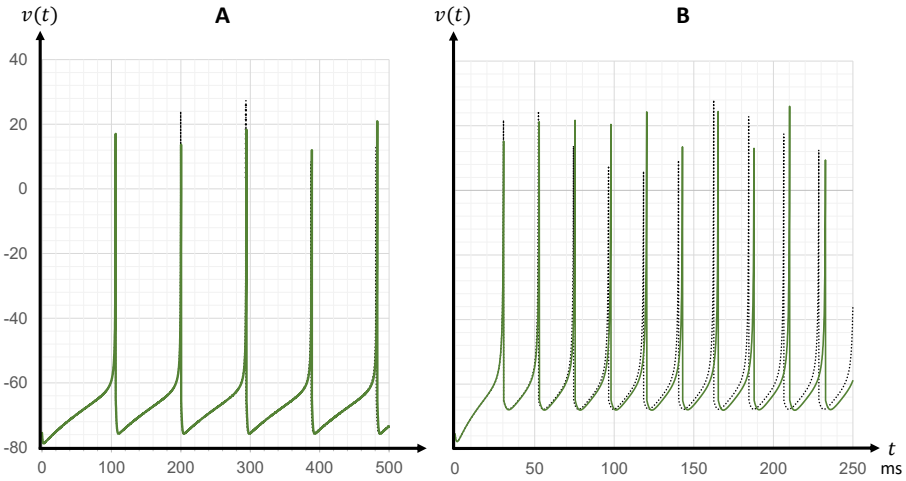


Figure 4.1 | Spike timing accuracy: comparison of the HNC node hardware ODE solver implementation with a reliable reference. Evolution of membrane potentials $v(t)$ recorded from: (**A**) a regular-spiking type; and (**B**) a fast-spiking type Izhikevich neuron. The neurons were stimulated with the constant current $i_{\text{ext}} = 5\,\text{pA}$. Neuron model dynamics were calculated using: the HNC node hardware implementation (green solid curves); and the GSL rkf45 ODE solver with an absolute integration error of 10^{-6} (black dotted curves), which is considered here as a reliable reference. A close match in the calculated dynamics is achieved for the regular-spiking type neuron. For the fast-spiking type neuron, a minimal delay in spike times is observed, which increases slightly over time.

non-stiff Izhikevich ODE system. Sufficient accuracy can be achieved with an explicit Forward Euler method – although it is the simplest numerical method available.

Nevertheless, the HNC node ODE solver implementation differs from the SpiNNaker variant. The ODE solvers of the HNC node use a fixed integration step size of $h = 0.1$ ms, which corresponds to the simulation resolution and the interval at which spike events are propagated. The SpiNNaker system instead propagates spike events at intervals of 1 ms, where ODE solvers need to perform sub-steps of $h = 1/16$ ms in order to achieve sufficient accuracy (see Listing 2.4).

In order to assess the accuracy of the HNC node ODE solver implementation, single neuron simulations were performed – analogously to the calculation verification task carried out in Section 2.3.4.2. The evolution of the membrane potentials were recorded and compared with a reliable reference, namely a software implementation of the models using a Runge-Kutta-Fehlberg(4, 5) (rkf45) ODE solver from the GNU Scientific Library (GSL)[1]. The results are shown in Figure 4.1. A close match in the calculated dynamics is achieved for the regular-spiking

[1] https://www.gnu.org/software/gsl/

type neuron (Figure 4.1A). The fast-spiking type neuron shows a small deviation, a delay in spike times, which increases slightly over time (Figure 4.1B). This result is consistent with the improved SpiNNaker implementation (see Figure 2.10), where this range of variation was found to be acceptable because the dynamics at the network level remain reproducible.

4.3 Implementation Verification

A common method to verify functionality and behavior of a logic design – i.e., verifying that it conforms to its functional specification – is simulation at the microarchitecture register-transfer level (RTL). For this purpose, the design to be verified is embedded in an HDL testbench; the design is then called *Design Under Test* (DUT). A testbench applies inputs to the DUT to verify whether correct outputs are produced. RTL simulation does not require a synthesis of the design. This is an advantage since the synthesis process is computationally intensive and time consuming. On the other hand, RTL simulation does not provide any information on whether timing constraints are met, and, when simulating a large design with complicated behavior, RTL simulation can be very time consuming too.

Here I propose a different and rather unusual approach to functional verification that takes advantage of the FPGA-SoC technology. Before it is explained in the next section, a note on the terminology in digital hardware development must be given: we need to distinguish between the terms *design* and *implementation*. While the term *design* refers to the textual HDL description, the latter is used to describe the processes of *translate*, *map*, and *place and route* performed by the synthesis tools to create a netlist. In the following, therefore, the use of the term *implementation* shall be understood as a synthesized design that can be loaded into an FPGA.

4.3.1 The Approach: *In-FPGA verification*

FPGAs powered by general-purpose processors have changed the way FPGAs are used. Here, hardware verification is rethought. For the functional verification of the HNC node's hardware components, an alternative strategy to an HDL testbench was used in which the target for verification is not the design, but its implementation. This approach exploits the Zynq-7000 SoC architecture to construct a fully software-driven testbench where the implementation to be verified, i.e., the DUT[2], is loaded into the FPGA of the SoC device.

With this *in-FPGA verification*, inputs to the DUT as well as the verification of its outputs, can be formulated in the C language, allowing complex verification tasks to be carried out

[2] In RTL simulation, *DUT* refers to a design. Here the term is also used for an implementation to be verified.

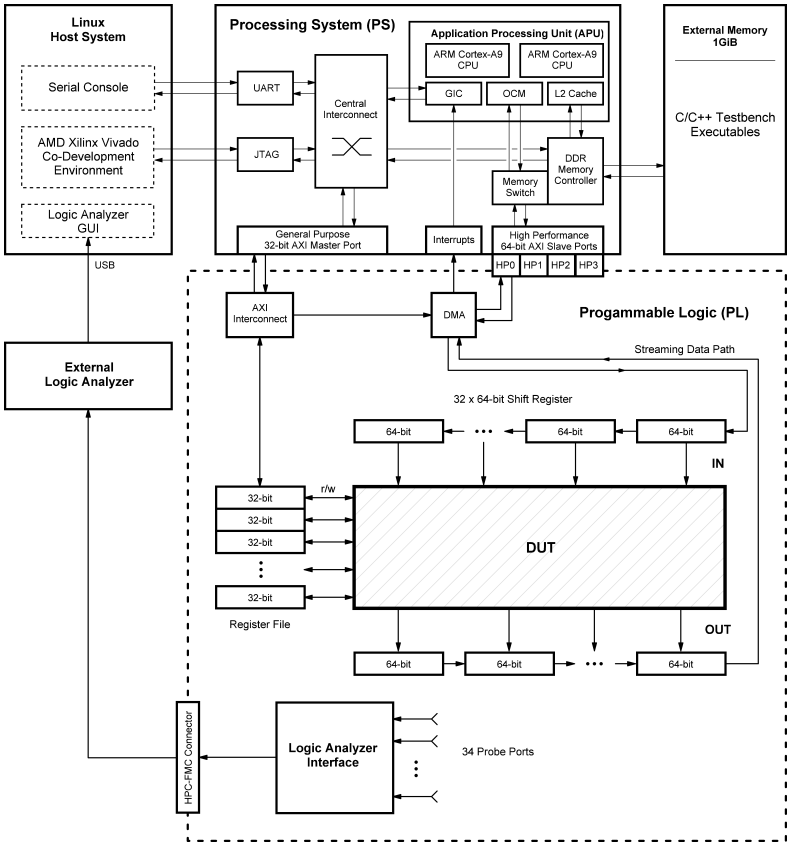


Figure 4.2 | Testbench. The testbench setup exploits the AMD Xilinx Zynq-7000 SoC device architecture. It makes use of the Application Processing Unit (APU) to set up a software-controlled test environment. The hardware component to be tested – the device under test (DUT) – can be connected to a register file as well as a streaming interface. For timing analysis and debugging purposes, a 34-channel logic analyzer interface is provided, which allows for probing any signal.

conveniently. This includes joint testing of the system's hardware and software components in the sense of functional hardware-software co-verification.

The testbench architecture as used for the functional verification of the HNC node is shown in Figure 4.2. Almost all hardware component function tests included this setup. The basic environment provides several supporting hardware blocks to which the DUT can be connected: a register file; a data stream interface; and a logic analyzer interface. A set of 16 32-bit registers

can be used to apply test-vectors and to read out internal states of the DUT. The register file is mapped into the APU's address space, and is thus accessible by the testbench software through simple memory read and write operations. Two chains of 32 x 64-bit shift-registers connected to a DMA soft-IP core build an interface to a streaming data path. This data path is used to put the DUT in a closed-loop operation with the APU to apply a stream of test-vectors. For timing analysis and debugging, the setup is complemented by a 34-channel logic analyzer interface[3].

During the development of the HNC node, the testbench was successively expanded with supporting logic, for example, for writing and reading on-chip BRAM memories and to trace content and utilization of FIFO buffers. These developments were accompanied by the development of the corresponding testbench software components.

4.3.2 Hierarchical Function Tests

Just as the development of the hardware and software components of the HNC node cannot be separated, neither can their verification be completely independent of each other. In the HNC node software system, higher-level functions build and depend on lower-level functions (see Figure 3.5). At the lowest level, the hardware abstraction layer, all routines have a hardware counterpart. While the high-level functions can be tested independently of the hardware blocks, this is not the case for low-level functions such as the DMA driver. Therefore, tests are built hierarchically from basic hardware and software tests to complex function, system and integration tests – just in the same way as it is common in pure software testing.

A comprehensive description of all component implementation tests and testbench setups would go beyond the scope of this thesis. Instead a selection and brief overview of the most relevant function tests and verification tasks conducted is given in Table 4.1, with the order of tests forming a hierarchy reflecting their dependencies.

4.4 Results Replicability

Simulation outcomes are expected to be replicable, that is, one expects spike-for-spike identical results in repeated simulations. To avoid that changes in the order of spike events result in changes in the accumulated synaptic inputs, the HNC node represents weights as fixed-point numbers. The additions performed in the RB pipelines are thus commutative. In this respect, spike order do not pose a problem for replicability. However, various reasons can be imagined that may also

[3]The interface was implemented for use with an Intronix LA1034 logic analyzer. (https://www.pctestinstruments.com/)

Component	Verification of	Description	Type of Test
AXI DMA soft-IP core	low-level driver and interrupt handler	• send and receive data packets in a closed-loop with the APU • perform pattern-tests • base test to also verify the function of the testbench	function test joint hw/sw basic
Data type conversion	correctness of fixed-point custom data type conversion	• perform basic mathematical operations with custom data types and verify results after backward conversion, for example: single float → s16.23 → add/sub/mult/div → single float	function test sw basic
ODE solver	main control FSM and ODE solver pipeline	• send and receive test data in a closed-loop with the APU • main control FSM is connected to the testbench register file to control operation and monitor states	function test hw advanced
ODE solver	accuracy of neuron and synapse model	• record membrane potentials and compares model dynamics to a reference, i.e., the results obtained from an rkf45 solver • requires data type conversion	calc. verif. test hw advanced
Neuron Manager	correctness of in-memory neuron instantiation	• generate SVB content in external memory and verify the correctness of the created structures • requires data type conversion	function test sw basic
SVB	SVB addressing and endianness conversion	• perform write and read operations using address-patterns to test correctness of addressing and endianness conversion (FPGA: 128-bit Big-Endian; APU: 128-bit Little-Endian)	function test hw basic
SVB	correctness of hw resources assignment	• perform a joint test of Neuron Manager and hw SVB to verify the assignment of neuron-ids to hw resources • requires data type conversion and the Neuron Manager	function test joint hw/sw advanced
Connection Manager	correctness of synapse instantiation	• generate synaptic target lists in memory and verify the correctness of the created structures • requires data type conversion	function test sw basic
PS/PL Data Transfer Module	read lists of synaptic targets from ext. memory	• initiate read operations from external memory to transfer synaptic target lists into FIFO buffers on the FPGA • requires data type conversion and the Connection Manager • testbench extension: read FIFO buffers	function test joint hw/sw advanced
RB and RB pipeline	RB pipeline and placement of events in RB	• inject defined spike events and read the content of RBs to verify the correctness of the content • requires data type conversion, the Connection Manager, and the PS/PL Data Transfer Module • testbench extension: software-programmable spike injection	function test joint hw/sw advanced
Serializer and Encoder	serialization of spike events and AER encoding	• apply test-vectors and verify the correctness of encoded • spike events collected in the FIFO buffer • testbench extension: read FIFO buffers	function test hw basic
Intra-node synchronization	synchronization of ODE solvers, RBs and spike communication	• system test to verify the interplay of components • uses a small test networks and analyzes spike events • requires all components	integration test joint hw/sw complex

Table 4.1 | Function tests. Listed is a selection showing the most relevant function tests and conducted verification tasks. The order of tests forms a hierarchy.

hinder replicability; for example, functional incorrectness of the RB algorithm implementation, or issues in the synchronization of hardware blocks.

To verify whether the HNC node produces replicable outcomes, spike recordings from repeated simulations of the two-population Izhikevich network were compared. The spike events were recorded from 20 minutes simulated time. To strengthen the test, neuron-ids were logically shifted

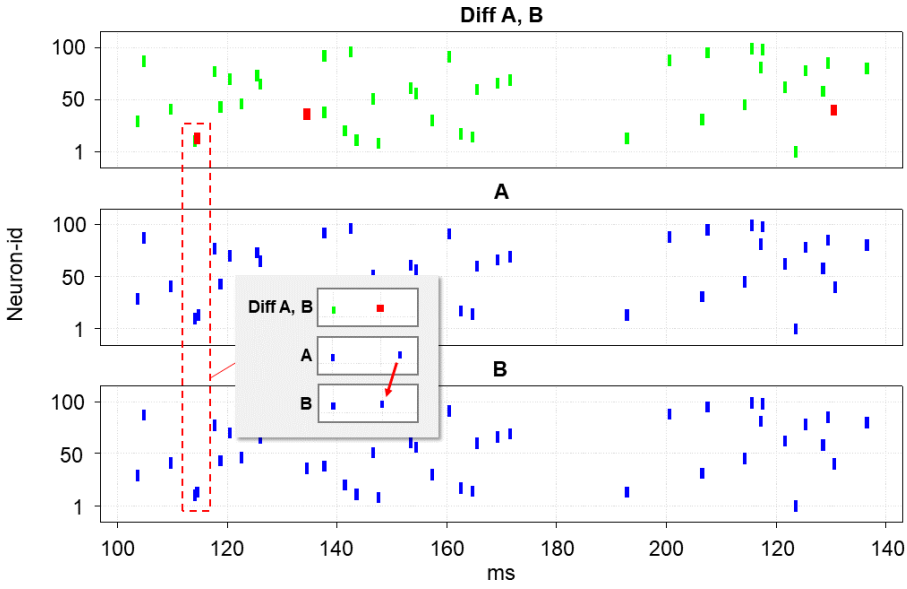


Figure 4.3 | Example output of the *spike-diff* utility. The *spike-diff* tool compares two data sets of spike recordings and visualizes their differences in spike raster plots. Shown is an example of a failed comparison. Differences in spike times and spike events are marked in red in the upper panel (**Diff A, B**), while matching events are indicated in green. Here, spike events from two simulation runs, recorded from the first 100 neurons, are compared within the interval 100 ms to 140 ms. Raster plots (**A**) and (**B**) display the spike events in each of the two data sets. Some spike events are shifted by one simulation time step, as shown in the inset and marked by the red arrow. Here, this difference in spike timing was caused by an intra-node synchronization issue, which led to an incorrect placement of spike events in the ring buffers.

from simulation run to simulation run. This procedure reassigned neuron-ids to different hardware resources, forcing a different spike order and scheduling of operations on each simulation run. For the data sets that were obtained from the multiple repetitions, a spike-for-spike comparison was performed.

In order to be able to carry out this comparison practically, a *spike-diff* utility was developed. It compares two data sets of spike recordings and detects differences in spike events and spike times. The tool allows the selection of a time window and a range of neurons, and visualizes the result in raster plots. An example of a failed test is given in Figure 4.3.

The replicability test uncovered several difficult-to-detect implementation issues. All of them were resolved, and the HNC node successfully passed the test.

4.5 Substantiation Assessment

In order to verify the HNC node's ability to meet the requirement for accurate and reproducible simulations, a model substantiation assessment, as introduced in Chapter 2, was performed. As test case network, the same two-population Izhikevich network model was used here. The assessment compared the dynamics of a selected network state obtained from a reliable reference C implementation of the network model with a reproduction of the model on the HNC node. To increase credibility, a second reference was included, namely a reproduction using the NEST simulator. For the comparison, the agreement between the three simulation outcomes was evaluated with respect to three selected statistical features of the network activity: firing rates, inter-spike intervals, and correlations.

4.5.1 Definition of the Substantiation Entities

The two-population Izhikevich network model has already been used in Section 2.3 to demonstrate a rigorous verification and validation workflow. For the definition and a description of the substantiation methodology entities *system of interest* and *mathematical model*, therefore, reference is made to Section 2.3.1 here. Furthermore, the different implementations of the Izhikevich network have also been subjected to various verification and validation tasks (see Section 2.3; and Gutzen et al., 2018; Pauli et al., 2018). The C and NEST executables can thus be considered as reliable references for defining the ground truth.

The substantiation assessment compared three *executable models*: (i) a first reference, an implementation in C; (ii) a second reference, an implementation in the NEST simulator; and (iii) the substantiation target, an instantiation of the Izhikevich network on the HNC node. In the following the three models are referred to as *C model*, *NEST model*, and *HNC node model*.

The technical conditions of the three models are different, so aspects of the simulation setup differ and require explanation.

C model

The model is constituted by the C implementation used throughout Section 2.3.4. This implementation has undergone a rigorous refactoring and a number of calculation verification tasks and is considered a reliable reference.

NEST model

To increase credibility, an implementation of the model in NEST is used as a second reference.

The model required a modification of the NEST source code and the built-in Izhikevich neuron model. The reason for this is as follows.

The Izhikevich network receives input from an external current source injecting an impulse into one randomly chosen neuron in each millisecond time step (see Section 2.3.1.1). In order to technically simplify the generation of such impulses, the NEST simulation was configured with a simulation resolution of 1 ms. This results in the ODE solver having to perform sub-steps to achieve sufficient accuracy. The C source code of the Izhikevich neuron model in NEST has been adapted such that the dynamics of the model progresses with a fixed integration step size of $h = 0.1$ ms, performing sub-steps with respect to the simulation resolution (see also Pauli et al., 2018). Note that spike events are communicated at intervals that correspond to the simulation resolution and forced to a grid point. The interval must be less or equal to the minimum synaptic delay in the network. This is given for the Izhikevich network, which has a minimum synaptic delay of 1 ms.

The NEST model was implemented using PyNEST[4]. Script and source code change are available on GitHub[5].

HNC node model

Network models on the HNC node are instantiated by a series of C-API `Create` and `Connect` calls, where an external C application generates the corresponding statements based on the model's connectivity data (e.g., exported from a PyNEST/NEST-generated network connectivity). Here, the connectivity of the Izhikevich model was exported from the C model, converted, and imported into the HNC node model. It should be noted that the HNC node model executable is formed by this instantiation as well as the HNC node hardware and software system.

All models use an explicit fixed-step size Forward Euler integration scheme with an integration step size of $h = 0.1$ ms, but differ in the following respects:

- While the HNC node model propagates spike events at 0.1 ms intervals, the C model and the NEST model use 1 ms intervals.
- The models use different random inputs to stimulate the Izhikevich network, which is a limitation caused by different PRNG implementations.

[4] PyNEST is a convenient Python interface to the NEST simulator.
[5] DOI: 10.5281/zenodo.6591552; `https://github.com/gtrensch/RigorousNeuralNetworkSimulations`

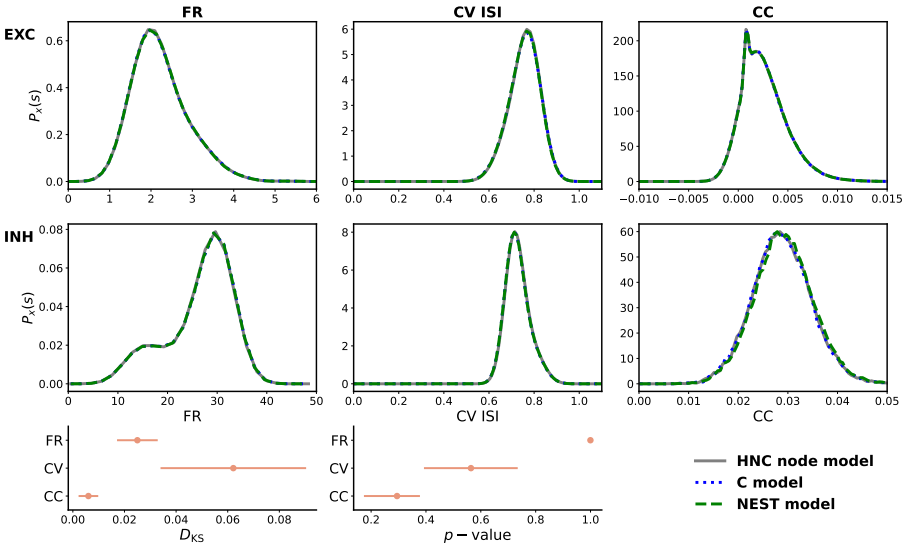


Figure 4.4 | Quantitative comparison of statistical measures. Upper two rows from left to right: probability distribution of average firing rate (FR), coefficient of variation (CV ISI), and Pearson's correlation coefficient (CC) for the excitatory (EXC) and the inhibitory (INH) population. The measures were calculated from 30 minutes simulated time. For the calculation of CC, spike trains were binned at 2 ms. The bottom row shows the Kolmogorov-Smirnov statistics calculated from the raw samples of the calculated statistical measures. All measures are in close agreement and show statistical equivalence.

4.5.2 Quantitative Comparison of Statistical Measures

In order to perform the assessment, a similar experimental setup was used as described in Section 2.3.2.1. Here, instead of employing five network states, only one network state is used – a decision justified by the absence of significant variations in the assessment across network states (see Section 2.3.2.1 and Appendix **B**).

Firstly, to create the reference network state, the ground truth, the C model was trained for one hour biological time using a spike-timing-dependent plasticity (STDP) rule. After one hour of simulated network time, the network connectivity was stored, that is, the initial condition of the network state. This connectivity data was then imported back into the C model. With the STDP rule deactivated, from 30 minutes simulated time, the spike events were recorded while the network was stimulated with a random input. This activity recording defined the ground truth – the dynamic state of the network after one hour of its evolution.

Secondly, the stored connectivity data was imported into the HNC node model. From 30 minutes simulated time, the spike events were recorded while the network was stimulated with

a different random input. To provide further evidence – to substantiate the correctness of the simulation result generated by the HNC node model – the procedure was repeated with the NEST model.

Finally, from the three obtained data sets of network activity, the probability distribution of the firing rates (FR), the coefficient of variation (CV ISI), and the Pearson's correlation coefficient (CC) were calculated and compared (see Section 2.3.2.2 for a description of the statistical measures). The result of the comparison is shown in Figure 4.4. For the calculation of CC, spike trains were binned at 2 ms. To derive the probability distributions from the calculated measures, the Freedman-Diaconis rule was applied to select the width of the bins of the distribution histograms. A Gaussian kernel was used for density smoothing. Also shown in Figure 4.4 (bottom row) is the Kolmogorov-Smirnov statistics calculated from the raw samples of the calculated statistical measures. The Kolmogorov-Smirnov test computes the supremum of the set of distances of two cumulative distribution functions, D_{KS}, and quantifies the discrepancy in the distributions by a *p-value*.

All measures are in close agreement and show statistical equivalence. Thus, we can conclude with a high degree of credibility that the implementation is correct.

4.6 Discussion

The design on RTL level in the VHDL language is very time consuming and error-prone. All the more attention must be paid to rigorous verification and validation. By following a systematic approach, rigor is brought to processes that build quality into implementations. The risk of implementation issues can be reduced and problems identified early in the design and implementation process. The verification and validation methods presented in Chapter 2, as well as the *in-FPGA verification* approach introduced, have proven to be very useful and expedient here.

Credibility of the correctness of the implementation

A high degree of credibility in the correctness of the implementation is created by the various verification tasks and the substantiation assessment that have been conducted. The selection of these activities was based on the work presented in Chapter 2.

Among the verification tasks, the replicability test is particularly noteworthy. It has proven to be extremely useful as it revealed issues on system-level that could not be detected by other functional tests; thus, the test complemented the simpler verification tests on component level.

The substantiation task was performed analogously to the worked example presented in Section 2.3, but with a few modifications to improve the quality of this assessment. The time

for which the network is simulated and network activity data is recorded has been increased from 5 minutes to 30 minutes. The longer spike trains allow better convergence of the pairwise Pearson's correlation coefficient (CC). The local coefficient of variation (LV) has been replaced by the standard coefficient of variation of inter-spike intervals (CV ISI). This is justified by the uniformity of the individual spike trains from which the measure is calculated; the oscillations that the network dynamics show are a population effect. For the comparison, the two populations of the network have been separated, and instead of histograms, the probability density functions of the measures were compared and their discrepancy quantified using a Kolmogorov-Smirnov test.

Although the models use different intervals at which spike events are communicated, the result of the assessment shows a very close agreement between them. However, one should consider that this *proof of correctness* of the HNC node implementation was demonstrated with an exemplary model. The assessment needs to be repeated as other neuron model types are added and algorithms change. Arguments for choosing the Izhikevich model have already been given in this work. Here, it should only be emphasized that the dynamics of the Izhikevich neuron model propagate numerical errors over time, which is not the case with other models where the model dynamics are reset after each spike event. In this respect, the Izhikevich neuron model places higher demands on the hardware implementation than, for example, a LIF neuron model.

In-FPGA verification

The *in-FPGA verification* approach has efficiently accompanied hardware-software co-development and well integrates with the AMD Xilinx Vivado tools. The software-driven approach has been effective for rapid creation of tests, test sequences, and even automation.

A central issue in design space exploration and the exploration of the FPGA-SoC technology was the achievable performance. Optimization was therefore a key development task. This refers to architecture, but also to the optimization of the designs in terms of latency. In this regard, the *in-FPGA verification* strategy allowed to accompany the development and testing of a component with an in-depth timing analysis. As a result, the critical timing-path that determines the maximum clock frequency of the HNC node design lies in a DSP slice – inaccessible for further optimization.

Moreover, a number of improvements in the HNC node design resulted from this approach. Several verification tests required additional hardware and software components to be added to the testbench in order to perform the tasks. Some of these extensions showed to be useful also in HNC node operation. Their functionality were therefore integrated into the prototype. These are in particular the following enhancements:

Software-programmable spike injection: As part of the functional verification of the *PS/PL Data Transfer Module* and the RB and RB pipelines, software-programmable spike injection was implemented to generate defined test-inputs. The functionality was integrated into the HNC node because it can also be used to provide a network with an external, software-generated stimulus.

Read and write access to FPGA memory resources by the APU: Originally implemented to manipulate and verify the content of on-chip BRAM memories and for convenient hardware debugging, it still serves these purposes. It allows access to FIFO buffers, RBs, and SVBs. Integrated into the HNC node, it is also used during node start up to bring all FPGA memory resources into a defined initial state. APU read and write access to RB and SVB memories is also a prerequisite for *checkpointing* functionality, i.e., suspending a simulation and saving its entire state so that a simulation can be restored and resumed at a later time. This is a feature frequently requested by users of software simulation tools.

Isolated operation of ODE solver pipelines in closed-loop with the APU: Technically, the correctness of the Izhikevich model and ODE solver implementation was verified by operating an ODE pipeline in a closed-loop setup with the APU. This turned out to be a very useful feature when implementing and testing new synapse and neuron models. Therefore, the functionality became a separate mode of operation of the HNC node.

The additional times that the *in-FPGA verification* approach require for design synthesis, i.e., *translate*, *map*, and *place and route*, have been found to be acceptable. A complete design run of the HNC node prototype takes approximately 15 minutes[6]. Synthesis time is significantly less for most hardware verification and development tasks since they typically do only include selected hardware modules – the amount of time it takes to run a design is highly dependent on the size of the design. The development and verification processes also benefit here from the modularity of the HNC node architecture design, i.e., the flexibility in the number of processing units included. In this manner, the HNC node was built with the RTL modules successively developed, tested, and integrated into the block design of the system.

[6]The time is observed for the AMD Xilinx Vivado 2019 toolchain running on an Ubuntu workstation equipped with an Intel(R) Core(TM) i7-7700K CPU 4.20 GHz.

Chapter 5

HNC Node Performance

"You can't always get what you want
But if you try sometimes you just might find
You get what you need"
The Rolling Stones

5.1 Introduction

Simulation tools like NEST (Gewaltig and Diesmann, 2007) are continually being developed to extend and improve their functionality. The developers of these tools invest considerable effort in optimizing codes, algorithms, and data structures to achieve the best possible performance and simulation efficiency (see, e.g., Kunkel et al., 2014; Pronold et al., 2022). These developments are accompanied by performance measurements to assess the effectiveness of code changes and to detect changes in runtime, scaling behavior, or resource utilization. These benchmarks are typically based on common network models.

One model used in the development of the NEST simulator, for example, is the balanced random network model described in Brunel (2000). The developers use it to perform strong-scaling[1] and weak-scaling[2] benchmarks on high performance computing (HPC) systems (see Kunkel et al., 2014). These benchmarks assess simulation real time factor (RTF)[3], network building time, simulation efficiency measured as energy per spike event, and resource utilization, i.e., memory consumption. By applying different sets of parameters that vary network size, connectivity, and number of processors (i.e., number of compute nodes, MPI processes, and threads), different workload situations are created. Conducting such benchmarks requires special attention to setup and parameterization in order to be comparable. For example, when performing weak-scaling benchmarks, the average firing rates in the network must be preserved; higher or lower firing rates will change the computational cost and skew the results. In this respect, the Brunel model is scalable, generating a computational cost that increases with network size.

Another example of a network model that has recently become popular as a reference for comparing architectures and even implementations on neuromorphic hardware is the cortical microcircuit model introduced by Potjans and Diesmann (2014) (see also Section 6.3 for a description of the model). It has been used in several studies to evaluate and demonstrate the performance of implementations and hardware systems, for example: in a performance comparison of the SpiNNaker neuromorphic system and the neural simulation tool NEST (van Albada et al., 2018); to evaluate the performance of simulation codes on GPUs (Golosio et al., 2021; Knight and Nowotny, 2018); to investigate the performance of the neural simulation tool NEST on a specific many-core hardware architecture (Kurth et al., 2022); to evaluate and demonstrate the performance of an implementation employing the IBM INC-3000 prototype FPGA-based neural computer (Heittmann et al., 2022); and to benchmark an FPGA-cluster comprising 35 NetFPGA SUME boards (Kauth et al., 2023).

[1] In strong-scaling benchmarks, the number of processors is increased while the problem size remains constant.
[2] In weak-scaling benchmarks, both the number of processors and the problem size are increased.
[3] The RTF is defined as wall clock time divided by simulated time and is the inverse of the acceleration factor.

A specific benchmark model provides a specific insight into a system's performance behavior; a node's workload depends on network size and network dynamics, which are tightly coupled to the model. A benchmarking approach that allows workloads to be varied irrespective of network size would be very valuable here, as it can provide a more comprehensive view of a system's performance characteristics.

In this chapter, such an approach is presented and used to systematically assess the performance characteristics of the HNC node. To this end, first, a workload model is developed that defines *workload* as the number of spike events per simulation time step, introducing a network size independent metric. Second, the knowledge about the HNC node microarchitecture and the latencies introduced by the components is exploited to derive an accurate performance model. Finally, the workload model and the performance model are then used to evaluate and predict the HNC node's behavior under varying workload situations and for different configurations and assumptions in design space. For the assessment, the minimal two-population Izhikevich network model is used as a *workload generator*.

Contributions

- A workload model is developed that defines a node's *workload* as the average number of spike events processed per simulation time step. For network sizes up to 10^5 neurons, the model introduces a metric that is independent of the number of neurons simulated.
- From the HNC node microarchitecture, an accurate performance model is derived. Although its formulation is specific to the HNC node, it captures in a general way the performance-determining aspects of the hybrid time- and event-driven scheme typically used in spiking neural network simulations of point neuron models.
- It is shown that by using a workload and performance model, the performance characteristics of the HNC node prototype can be predicted as a stand-alone compute node, as well as when operating in a cluster and under varying assumptions regarding workload and hardware design choices. This allows bottlenecks to be identified and future developments to be guided.

5.2 Methods and Materials

5.2.1 Workload Model

The number of neurons an HNC node can simulate is a hardware design parameter and is fixed. Therefore, the computational cost of advancing neuron states does not vary between simulation

time steps, introducing a constant latency to the processing. In contrast, the computational cost of processing spike events varies and depends on the firing statistics of neurons and their connectivity. It introduces a latency to operation that depends on the number of spike events arriving at a node in a simulation time step, as well as the number of synapses on the node that connect the firing neurons to their postsynaptic neurons. It is thus determined by the number of presynaptic data items to be processed, that is, the number of synapse updates performed. The above can be used to derive a convenient metric for a node's *workload*.

For a given number of neurons per node N^{M}, a certain number of nodes M is required to simulate a network of size N. The connection probability ϵ of the network (here assumed to be uniform across the network) determines the average in/out-degree $\bar{K}_{\mathrm{in/out}} = \epsilon N$, i.e., the number of in- and outgoing synaptic connections of a neuron. The in/out-degree $\bar{K}_{\mathrm{in/out}}$ grows with the network size, but has an upper limit. A typical cortical neuron connects to between 10^3 and 10^4 other neurons. Given a connection probability value of approximately $\epsilon = 0.1$ observed in Braitenberg and Schüz (1998) for neural connectivity, a network of 10^5 neurons, where each neuron forms 10^4 connections, represents this upper limit. For network sizes up to this scale, we can define a metric that is independent of the number of neurons simulated.

Assuming uniform network connectivity and evenly distributed connections across nodes, the average number of postsynaptic targets per presynaptic neuron per node is given by the mean out-degree divided by the number of nodes. This can be written as

$$\bar{C}^{\mathrm{M}} = \begin{cases} \epsilon N^{\mathrm{M}} & \text{if } N \leq 10^5 \\ 10^4 \, \dfrac{N^{\mathrm{M}}}{N} & \text{otherwise.} \end{cases} \tag{5.1}$$

For $N \leq 10^5$, $\bar{C}^{\mathrm{M}}$ does not depend on the network size. It is a constant determined by the number of neurons per node N^{M} and the connection probability ϵ in the network. Thus, the number of synapse updates to be performed on a node per spike event is also constant on average. It is therefore practical to consider as an indicator of a node's *workload* the average number of spike events processed per simulation time step k, which can be formulated as

$$\bar{\nu}_{\mathrm{k}} = N \bar{\nu} h, \tag{5.2}$$

where $\bar{\nu}$ denotes the average firing rate calculated over all neurons in the network by

$$\bar{\nu} = \frac{1}{N} \sum_{i=1}^{N} \frac{n_i^{\mathrm{sp}}(T)}{T}. \tag{5.3}$$

In Equation (5.3), $n_i^{\mathrm{sp}}(T)$ is the total spike count of neuron i in the interval T, and h defines the temporal resolution of the simulation. From the Equations (5.2) and (5.3), and the definition of the firing rate (Equation (2.5)), we derive

$$\bar{\nu}_{\mathrm{k}} = h \sum_{i=1}^{N} \mathrm{FR}_i, \tag{5.4}$$

where FR_i is the firing rate of neuron i.

Up to a network size of 10^5 neurons, we can use this metric to simulate the workload of a node regardless of the *real* network size: from a single node's perspective, the workload that a small network of neurons with high average firing rates generates is equivalent to a large network where neurons have low firing rates; the node *sees* the same workload.

Since the number of synapses formed by a neuron cannot become arbitrarily large, $\bar{C}^{\mathrm{M}}$ does not remain constant beyond a certain network size. When the network has reached a natural density, $\bar{C}^{\mathrm{M}}$ decays with the network size, i.e., with the number of nodes $M = N/N^{\mathrm{M}}$. In this respect, and from the perspective of a node, a network of approximately 10^5 neurons represents an upper bound on the workload generated: while the number of incoming spike events increases with network size, the number of synapse updates per spike event and node decreases accordingly.

5.2.2 Performance Model

In Section 3.3.5, the operating latencies have been extracted from the HNC node microarchitecture. This knowledge is exploited in the following to derive a performance model that allows conclusions to be drawn about the performance characteristics of the HNC node, its behavior under different workload situations, and with regard to design and technology parameters. The workload metric that was introduced in the previous section will be used here.

Single node

The time span to perform a single simulation step becomes minimal if no spike event occurs, and is predominantly determined by the number of serially processed neurons assigned to an ODE solver pipeline. This is reflected in the ODE solver pipeline iteration latency $L_{\mathrm{ODE,IL}}$. Together with the synchronization latency L_{SYNC}, it sets the upper bound for the single-node acceleration factor $F_{\mathrm{S}}^{\mathrm{MAX}}$ at a given clock frequency f_{clk}. From the timing diagram in Figure 3.24A, we derive

$$F_{\mathrm{S}}^{\mathrm{MAX}} = \frac{khf_{\mathrm{clk}}}{L_{\mathrm{RD}} + L_{\mathrm{ODE,LL}} + k(L_{\mathrm{ODE,IL}} + L_{\mathrm{SYNC}})}, \tag{5.5}$$

where k denotes the number of simulation steps, and h specifies the temporal resolution of the simulation. For $k \gg 1$ this simplifies to

$$F_{\mathrm{S}}^{\mathrm{MAX}} = \frac{h f_{\mathrm{clk}}}{L_{\Sigma}} \quad \text{with} \quad L_{\Sigma} = L_{\mathrm{ODE,IL}} + L_{\mathrm{SYNC}}. \tag{5.6}$$

In Equation (5.6), L_{Σ} summarizes the processing latencies for the non-spiking case. Similarly, latencies from processing spike events can be summarized according to the timing diagrams and process scheduling shown in Figure 3.24A and Figure 3.24B. This sum includes the latencies of spike events serialization and buffering ($L_{\mathrm{SE}} = L_{\mathrm{SEP}} + L_{\mathrm{SES}} + L_{\mathrm{SEF}}$), the latencies incurred by the initiation of the data streams S1 and S2 ($L_{\mathrm{IS}} = L_{\mathrm{ISCAL}} + L_{\mathrm{ISADR}}$), and the latencies resulting from the RB processing of pending data items at the end of a simulation time step ($L_{\mathrm{RB}} = L_{\mathrm{RBF}} + L_{\mathrm{RBP}}$). The number of clock cycles of each of the latencies as well as a description, can be found in Figure 3.24C. Altogether, this results in

$$L_{\Sigma}^{\mathrm{SE}} = L_{\mathrm{RD}} + L_{\mathrm{ODE,LL}} + \frac{L_{\mathrm{ODE,IL}}}{2} + L_{\mathrm{SE}} + L_{\mathrm{IS}} + L_{\mathrm{RB}} + L_{\mathrm{SYNC}}^{\mathrm{SE}}. \tag{5.7}$$

The term $L_{\mathrm{ODE,IL}}/2$ in Equation (5.7) expresses the assumption that, on average, spike events occur in the middle of an ODE solver pipeline iteration – a simplification justified here by the round-robin assignment of neurons to processing units and pipelines, ensuring an even distribution of spike events across the clock cycles during pipeline operation.

For an isolated node with no inter-node communication, the acceleration factor as a function of the *workload* $\bar{\nu}_{\mathrm{k}}$, can then be formulated as

$$F_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}}) = \begin{cases} \dfrac{h f_{\mathrm{clk}}}{\bar{\nu}_{\mathrm{k}}(L_{\Sigma}^{\mathrm{SE}} + L_{\mathrm{S}}) + (1 - \bar{\nu}_{\mathrm{k}}) L_{\Sigma}} & \text{if } \bar{\nu}_{\mathrm{k}} < 1 \\ \dfrac{h f_{\mathrm{clk}}}{L_{\Sigma}^{\mathrm{SE}} + \bar{\nu}_{\mathrm{k}} L_{\mathrm{S}}} & \text{otherwise.} \end{cases} \tag{5.8}$$

For workloads $\bar{\nu}_{\mathrm{k}} < 1$, the denominator in Equation (5.8) consists of two terms corresponding to the spiking ($\bar{\nu}_{\mathrm{k}}(L_{\Sigma}^{\mathrm{SE}} + L_{\mathrm{S}})$) and the non-spiking ($(1 - \bar{\nu}_{\mathrm{k}})L_{\Sigma}$) case, where L_{S} denotes the per spike event data stream latency, i.e., the number of clock cycles required to retrieve a single list of synaptic targets (LST). The two branches are equal if $\bar{\nu}_{\mathrm{k}} = 1$. In the absence of spike events, $F_{\mathrm{S}}(0) = F_{S}^{\mathrm{MAX}}$ applies (Equation (5.6)). Note that Equation (5.8) does not consider $\bar{C}^{\mathrm{M}}$, the average number of postsynaptic targets per presynaptic neuron per node. The value of $\bar{C}^{\mathrm{M}}$ determines the value of L_{S}, which cannot be derived from the microarchitecture and was therefore measured. Furthermore, it is ignored that a data transfer could be completed before all neurons

have been processed, i.e., $L_{\mathrm{ODE,IL}}/2 - L_{\mathrm{S}} > 0$. To account for this, the value of L_{Σ}^{SE} would have to be corrected by adding the latency $L_{\mathrm{ODE,IL}}/2 - L_{\mathrm{S}}$. However, an effect can only be seen at very low spike rates, because it holds: $\bar{\nu}_{\mathrm{k}} L_{\mathrm{S}} \gg L_{\mathrm{ODE,IL}}/2 - L_{\mathrm{S}}$. Equation (5.8) therefore allows for the calculation of a good estimate of the acceleration factors achievable at different workloads.

Cluster node

By extending the model to include inter-node communication latencies, the performance characteristics of a multi-node system can also be described. Note, that the model can only provide an estimate here, as communication latencies cannot be derived from the single-node prototype. Strongly simplifying the complex effects of communication network topologies and protocols, three basic assumptions are made:

- Spike events are broadcast, i.e., communicated to all nodes.
- Inter-node connections all have the same and fixed transmission latency time T_{COM}, which adds to every simulation step.
  In addition to the times needed to communicate the spike events, T_{COM} also includes inter-node synchronization latency.
- Every spike event adds an additional latency to communication.
  This takes into account that inter-node communication times increase with workload. This latency should be defined as $\bar{\nu}_{\mathrm{k}} \alpha T_{\mathrm{COM}}$, where αT_{COM} specifies a small fraction of the transmission latency time, i.e., $\alpha < 1$.

Extending the Equation (5.8) accordingly results in

$$F_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}}) = \begin{cases} \dfrac{h f_{clk}}{\bar{\nu}_{\mathrm{k}}(L_{\Sigma}^{\mathrm{SE}} + L_{\mathrm{S}} + \alpha L_{\mathrm{COM}}) + (1 - \bar{\nu}_{\mathrm{k}}) L_{\Sigma} + L_{\mathrm{COM}}} & \text{if } \bar{\nu}_{\mathrm{k}} < 1 \\ \dfrac{h f_{clk}}{L_{\Sigma}^{\mathrm{SE}} + \bar{\nu}_{\mathrm{k}}(L_{\mathrm{S}} + \alpha L_{\mathrm{COM}}) + L_{\mathrm{COM}}} & \text{otherwise,} \end{cases} \tag{5.9}$$

where L_{COM} denotes the transmission latency in clock cycles derived from the transmission latency time, i.e., $L_{\mathrm{COM}} = f_{\mathrm{clk}} T_{\mathrm{COM}}$. Note that L_{COM} in Equation (5.9) does not vanish even in the absence of spike events. This takes into account inter-node synchronization times. According to Equation (5.5), the upper bound for the acceleration factor with inter-node communication then becomes

$$F_{\mathrm{C}}^{\mathrm{MAX}} = \frac{h f_{\mathrm{clk}}}{L_{\Sigma} + L_{\mathrm{COM}}}. \tag{5.10}$$

For a performance analysis it is convenient to calculate the total relative performance loss in percent with respect to the maximum achievable acceleration factor. It can be estimated for different workloads as

$$P_{\mathrm{TOT}}(\bar{\nu}_{\mathrm{k}}) = P_{\mathrm{S}} + P_{\mathrm{C}} = \left(1 - \frac{F_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}})}{F_{\mathrm{S}}^{\mathrm{MAX}}}\right) \cdot 100\%. \tag{5.11}$$

This can be further subdivided into the loss caused by the processing of spike events

$$P_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}}) = \left(1 - \frac{F_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}})}{F_{\mathrm{S}}^{\mathrm{MAX}}}\right) \cdot 100\%, \tag{5.12}$$

and the loss caused by inter-node communication

$$P_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}}) = \frac{F_{\mathrm{S}}(\bar{\nu}_{\mathrm{k}}) - F_{\mathrm{C}}(\bar{\nu}_{\mathrm{k}})}{F_{\mathrm{S}}^{\mathrm{MAX}}} \cdot 100\%. \tag{5.13}$$

5.3 Results

Several tasks were performed to systematically assess the performance characteristics of the HNC node. First, the achievable acceleration factors under varying workload situations were measured. The two-population Izhikevich network model (see Section 2.3.1.1) was used here as a *workload generator*. As a reference and benchmark, the results were compared with the acceleration factors achieved in comparable simulations performed with the neural simulation tool NEST. Second, in order to verify the correctness of the HNC node performance model, the measured acceleration factors were compared with the acceleration factors predicted by the model. Finally, the performance model was then used to evaluate the performance characteristics of the HNC node as a stand-alone compute node and when operating in a cluster, examining the effect of additional synchronization and communication latencies, and under varying assumptions regarding workload and hardware design choices. To this end, four different sets of hardware design parameters were selected for an investigation.

5.3.1 Single Node Performance

To measure the acceleration factors under varying workload situations, the two-population Izhikevich network was run multiple times, with each run creating a different workload situation. In simulations of 300 s simulated time, the network was stimulated with an external input current, i_{ext}. From simulation run to simulation run, i_{ext} was systematically varied from $i_{\mathrm{ext}} = -3.0$ pA to 100.0 pA. This caused the network to run through a wide range of activity, from quiescence up to

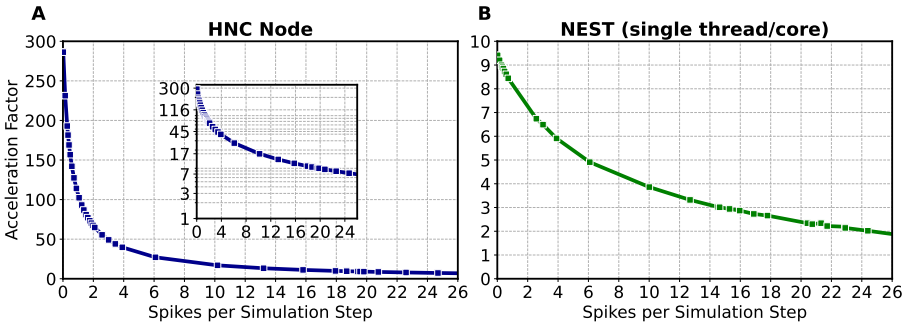


Figure 5.1 | Acceleration factors achieved by the HNC node and the simulation tool NEST. Acceleration factor (simulated time divided by wall clock time) achieved under different workloads (spike events per simulation time step) for: (**A**) the HNC node at a PL clock frequency of $f_{\text{clk}} = 200$ MHz; and (**B**) the neural simulation tool NEST on an Intel(R) Core(TM) i7-7700K CPU 4.20 GHz (Kaby Lake architecture). The inset in (**A**) gives a log-lin representation.

an average firing rate of $\bar{\nu} = 300$ spks/s. According to the workload model (Equation (5.4)), this resulted in an average number of spike events per simulation time step of $\bar{\nu}_{\text{k}} = \{0, \ldots, 30\}$ spks/h. In this way, an increasing workload was generated and different workload scenarios were simulated for the HNC node. The durations of the simulations were measured and the acceleration factors were calculated as the quotient of the simulated time and the measured physical wall clock time. The result of these measurements is shown in Figure 5.1A.

For comparison, the simulations were repeated using the simulation tool NEST. For this purpose, the model was implemented in NEST 2.20.1 (Fardet et al., 2020). The simulations were performed on a desktop computer powered by an Intel(R) Core(TM) i7-7700K CPU 4.20 GHz (Kaby Lake architecture). The measured acceleration factors are shown Figure 5.1B.

For low workloads, up to a few spike events per simulation time step, the HNC node outperforms the NEST simulations on the Intel Kaby Lake CPU. A low workload is generated, for example, when the Izhikevich network is not stimulated with an additional external offset current ($i_{\text{ext}} = 0$ pA). The average firing rate in the network is then $\bar{\nu} = 7.0$ spks/s, which corresponds to an average number of spike events processed per simulation time step of $\bar{\nu}_{\text{k}} = 0.7$ spks/h. For this workload, the acceleration factor achieved by the NEST simulator is 8.4, whereas the HNC node reaches a factor of 127. Here, the HNC node can take full advantage of hardware parallelism, hardware pipelining, and data locality. As the workload increases, the HNC node experiences rapid performance deterioration. Spike processing shifts the cost of computation. Data access latency and the limited bandwidth of the external memory decelerate processing. A similar behavior can be observed for NEST on the CPU, although not as pronounced since the base

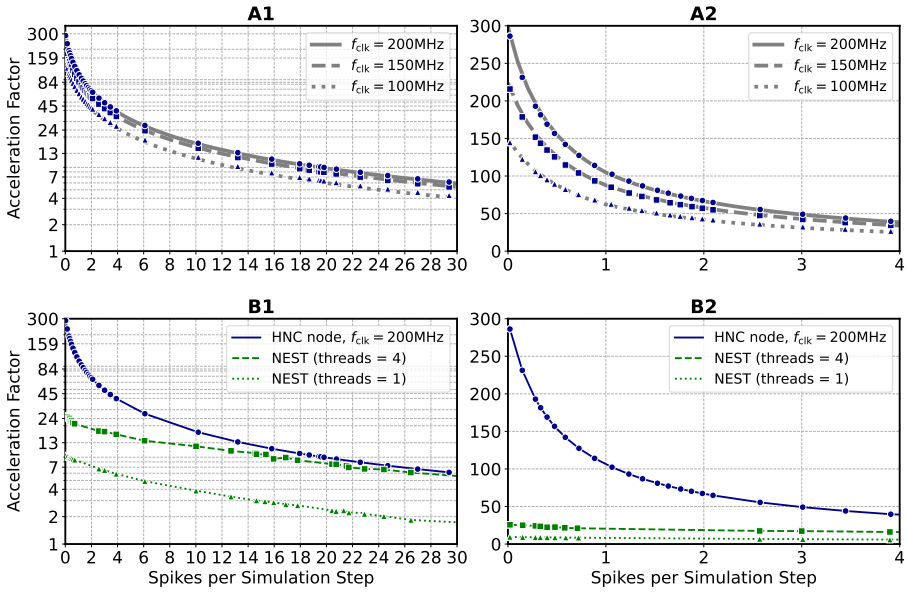


Figure 5.2 | Acceleration factors achieved by the HNC node at different PL clock frequencies and in comparison with the simulation tool NEST. (**A**) Acceleration factors achieved by the HNC node (blue markers) as a function of workload and at three different PL clock frequencies in log-lin (**A1**) and linear (**A2**) representation. The gray curves show the acceleration factors predicted by the performance model. (**B**) as in (**A**), but compares the acceleration factors achieved by the HNC node running at a PL clock frequency of f_{clk} = 200 MHz with those achieved by a NEST simulation of the same model when using one or four threads of an Intel(R) Core(TM) i7-7700K CPU 4.20 GHz.

acceleration is much lower.

The NEST simulator is a runtime-optimized, flexible tool for neural network simulations and represents a good reference in this regard. Clearly, a CPU-optimized implementation of the specific network model could achieve even better results[4]. However, the difference in performance and efficiency is such that the HNC node performance is beyond the reach of any CPU implementation.

In order to verify the correctness of the HNC node performance model, the performance measurements carried out on the HNC node were repeated at the three different PL clock frequencies: 100 MHz; 150 MHz; and 200 MHz. The measured acceleration factors were then compared with the acceleration factors predicted by the performance model using Equation (5.8). This

[4] A C implementation of the network model is provided on GitHub: https://github.com/gtrensch/RigorousNeuralNetworkSimulations

comparison is shown in the upper panels, A1 and A2, of Figure 5.2. The predicted acceleration factors are in almost perfect agreement with the measured values.

Memory access latency is a major performance determining factor. In Equation (5.8), this is expressed by the term $\bar{\nu}_k L_S$, where L_S is the number of clock cycles required to read an LST from external memory. This latency cannot be compensated by a higher PL clock frequency. However, an acceleration factor of approximately 100 is achieved for moderate workloads, i.e., $\bar{\nu}_k \approx 1$ spks/h. Such a workload is created, for example, by a network consisting of $N = 5000$ neurons with an average firing rate of $\bar{\nu} \approx 2$ spks/s.

Also shown in Figure 5.2 (lower panels, B1 and B2) is a comparison of the acceleration factors measured for the HNC node at a PL clock frequency of $f_{clk} = 200$ MHz with equivalent simulations in NEST performed on a quad-core Intel CPU. Even at high workloads, a single HNC node performs substantially better than a single state-of-the-art processor core.

The high workloads generated here are realistic scenarios in large-scale simulations and are not only of theoretical interest for benchmarking tasks. For example, in simulations of the cortical microcircuit model (Potjans and Diesmann, 2014), which consists of $N \approx 0.8 \cdot 10^5$ neurons, a value of $\bar{\nu}_k \approx 25$ spks/h can be observed (see Section 6.3). At this workload the HNC node achieves an acceleration factor of approximately 7, whereas for a single-threaded NEST simulation a factor of approximately 2 was measured. When distributing workload using all four cores of the Intel CPU, the NEST simulation is nearly as fast as a single HNC node. Note that this is derived from the performance model. The single HNC node prototype cannot accommodate a network as large as the cortical microcircuit model.

Although power efficiency is not a central theme of this work, it is worth noting that the power consumption of the HNC node SoC device is on the order of a few watts (see the power report in Section 3.3.6) and thus achieves a much higher simulation efficiency than the Intel core, for which a power consumption of several tens of watts is to be expected.

5.3.2 Performance Characteristics for Different Sets of Design Parameters

The design space is complex, and it is not always obvious what the appropriate design choices are. In a high workload scenario, reducing the number of processing units may improve simulation efficiency even though it increases the number of neurons processed by a processing unit (given a fixed number of neurons per node). This may seem like a contradiction, but the reasoning becomes apparent when analyzing the performance behavior for varying workloads. In a cluster, the adverse effect that inter-node communication has on performance also needs to be considered in finding an optimal node setup.

To illustrate this, the HNC node's performance characteristics are examined for the following four different sets of design parameters:

- *Prototype*
  This set of parameters corresponds to the prototypical HNC node implementation with which the measurements have been carried out. The parameter set defines: $P = 16$ processing units; $N^{\mathrm{P}} = 64$ neurons per processing unit; and $S = 2$ data streams (see Figures 3.11 and 3.12).

- *High data stream parallelism*
  Similar to the *prototype* parameter set, but assumes that each processing unit is assigned a data stream, providing an eightfold reduction in external memory access latency. The parameter set defines: $P = 16$; $N^{\mathrm{P}} = 64$; $S = 16$.

- *High processing units parallelism*
  As for the *high data stream parallelism* parameter set, but implementing twice the number of processing units in order to halve the ODE solver pipeline latency $L_{\mathrm{ODE,IL}}$, and increase the maximum achievable single-node acceleration factor. The parameter set defines: $P = 32$; $N^{\mathrm{P}} = 32$; $S = 16$.

- *Low processing units parallelism*
  The parameter set implements the opposite of *High processing units parallelism* reducing the number of processing units. The parameter set defines: $P = 8$; $N^{\mathrm{P}} = 128$; $S = 16$.

The number of neurons per node ($N^{\mathrm{M}} = PN^{\mathrm{P}}$) is the same for all parameter sets. Note that the parameter sets, with the exception of the *prototype* configuration, have not been implemented and applied to the HNC node. The selected SoC device is here limited to the *prototype* configuration in terms of the number of data streams, i.e., the external memory bandwidth. In the following, the performance model introduced in Section 5.2.2 is used to derive estimates of the performance characteristics.

To describe the effect of inter-node communication on performance, the performance model introduces the two parameters: transmission latency time T_{COM}; and a per spike event transmission latency factor α (for a description of the parameters see Section 5.2.2). Their values were set to $T_{\mathrm{COM}} = 500\,\mathrm{ns}$ and $\alpha = 0.05$. They are the same for all parameter sets. The choice for the transmission latency time is motivated by the temporal resolution of $h = 0.1\,\mathrm{ms}$ and an envisioned acceleration factor of 100, which would be a major breakthrough. This assigns T_{COM} half of the wall clock time that would be available to complete a single simulation step. The value of

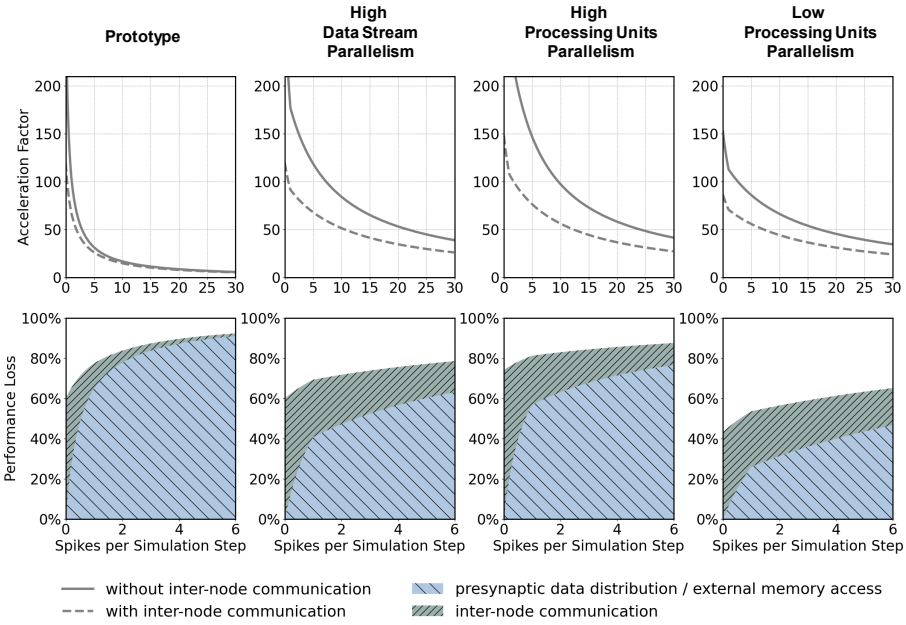


Figure 5.3 | Performance characteristics estimation. Performance characteristics of the HNC node calculated using the performance model (Section 5.2.2) for the parameter sets *prototype*; *high data stream parallelism*; *high processing units parallelism*; *low processing units parallelism*. See main text and Table 5.1 for a description of the parameter sets. The upper panels show the achievable acceleration factors as a function of workload with inter-node communication $F_C(\bar{\nu}_k)$ (dashed curves) and without inter-node communication $F_S(\bar{\nu}_k)$ (solid curves); the lower panels show the stacked plots of the respective contributions to the loss of performance with respect to the maximum achievable single-node acceleration factor F_S^{MAX} of the inter-node communication $P_C(\bar{\nu}_k)$ (green) and presynaptic data distribution $P_S(\bar{\nu}_k)$ (blue).

the per spike event transmission latency factor α was arbitrarily chosen and corresponds to five additional clock cycles per spike event at the PL clock frequency $f_{clk} = 200$ MHz.

The estimated performance characteristics for the four parameter sets are shown in Figure 5.3. The upper panels displays the acceleration factors with and without inter-node communication as a function of workload, calculated using the Equations (5.8) and (5.9). The lower panels provide an alternative view, illustrating the respective proportion of the performance loss in percent with respect to the maximum achievable single-node acceleration factor for the corresponding parameter set. Shown are the performance losses caused by inter-node communication, and by the process of presynaptic data distribution, which includes external memory access times. These characteristics are derived from the Equations (5.12) and (5.13) of the performance model.

Parameter	Prototype	High Data Stream Parallelism	High Processing Units Parallelism	Low Processing Units Parallelism
Number of parallel data streams, S	2	16	16	16
Data stream latency, L_S	110	14	14	14
Number of processing units, P	16	16	32	8
Number of neurons per processing unit, N^P	64	64	32	128
ODE pipeline iteration latency, $L_{ODE,IL}$	64	64	32	128
Acceleration Factors w/o Communication (Single Node)				
Maximum, $F_S^{MAX} = F_S(\bar{\nu}_k = 0)$	298.5	298.5	571.4	152.7
Low workload, $F_S(1.0)$	104.7	177.0	246.9	113.0
Medium workload, $F_S(10.0)$	16.9	84.5	97.7	66.5
High workload, $F_S(20.0)$	8.8	52.4	58.4	45.6
Acceleration Factors with Communication (Cluster)				
Maximum, $F_C^{MAX} = F_C(0)$	119.8	119.8	148.1	86.6
Low workload, $F_C(1.0)$	67.6	91.7	107.5	70.9
Medium workload, $F_C(10.0)$	15.0	51.7	56.4	44.4
High workload, $F_C(20.0)$	8.1	34.8	36.9	31.3

Table 5.1 | Acceleration factors for four different parameter sets. Listed are the achievable acceleration factors for four different parameter sets: *prototype*; *high data stream parallelism*; *high processing units parallelism*; *low processing units parallelism*, and for three different workload situations: *low*; *medium*; *high* as well as with and without inter-node communication. The number of neurons per node $N^M = PN^P = 1024$, the PL clock frequency $f_{clk} = 200$ MHz, the transmission latency time $T_{COM} = 500$ ns, and the per spike event transmission latency factor $\alpha = 0.05$ (see main text) are the same for all parameter sets. The acceleration factors have been calculated using the performance model described in Section 5.2.2.

Table 5.1 lists the calculated acceleration factors achieved by the different setups for low, medium, and high workload.

It requires no explanation that inter-node communication latency de facto reduces the maximum achievable acceleration factors. For the *prototype* configuration (Figure 5.3, prototype, upper panel), for example, the acceleration factor decreases from 298.5 to 119.8 (Table 5.1). As the workload increases, the effect becomes progressively smaller. For low workload, the factor decreases by 35.5%, for medium workload by 11.2%, and for high workload by 7.9%. For low workload, the achievable acceleration is determined by inter-node communication latency, but toward higher workload external memory access time is the main contributor to performance degradation (Figure 5.3, prototype, lower panel).

In the *high data streaming parallelism* configuration, each processing unit has a data stream assigned, and by this means, an eight times higher parallelism in the presynaptic data distribution is introduced; the two data streams S1 and S2 (Figure 3.11) are each split into eight streams, thus reducing external memory access times by a factor of eight. Figure 5.3 (high data stream parallelism, upper and lower panel) illustrates the effect. For medium workload and with inter-

node communication, the acceleration factor increases from 15.0 (for the prototype configuration) to 51.7, i.e., by a factor of 3.4.

We can try to further improve performance by increasing the parallelism in the computation of the model dynamics by adding processing units. The *high processing units parallelism* configuration doubles the number of processing units. This configuration achieves a very high maximum acceleration factor of 571.4 for the single node without inter-node communication. In a cluster such high acceleration cannot be realized, even for low workload. Bound by inter-node communication latency, the performance loss in relation to the maximum acceleration is 74%, and for low workload 81.2%. However, for high workload, external memory access time is still the main limiting factor (Figure 5.3, high processing units parallelism, upper and lower panel). In addition, this setup doubles the hardware footprint of the ODE solver pipelines. Minimizing the hardware footprint is always a concern in a design as it reduces power consumption.

The *low processing units parallelism* configuration implements half of the processing units of the *prototype* configuration (Figure 5.3, low processing units parallelism, upper and lower panel). For low workload, and in comparison with the *high processing units parallelism* configuration, the acceleration factor decreases from 107.5 to 70.9, i.e., by 34%. For high workload, the acceleration factor decreases from 36.9 to 31.3. This is a loss of 15.2% and may be an acceptable degradation when making design decisions oriented toward a high workload scenario, considering that it can save 75% of ODE solver pipeline hardware resources, namely DSP units.

Depending on the workload being processed and the design goal, different setups may be appropriate, with available memory bandwidth also determining the degree of reasonable processing parallelism.

5.4 Discussion

The HNC node is designed to operate in a cluster. Here, the workload and performance model can predict the performance characteristics and enable the study of the performance behavior under varying workload situations and for different sets of design parameters. This is of value as it provides guidance for design decisions and future development, as well as the ability to identify bottlenecks and pinpoint areas of constraint.

Workload model and workload generation

The proposed workload model has proven to be very practical. Since the introduced metric is independent of network size, large workloads can be *simulated* and scenarios that only occur in simulations of large networks can be studied in a small setup. By using the minimal two-

population Izhikevich network as a *workload generator*, a wide range of workload situations can be created. This also includes unrealistically high spike rates, which have found practical use in system verification; for example, in a stress test where the HNC node's spike communication and recording hardware is pushed to its limits to verify that no spike losses occur under high workloads.

Nonetheless, the Izhikevich network is not an ideal choice to be used in such a way. Attempts to scale the network to several thousand neurons to perform measurements on traditional multi-node systems failed. Network dynamics become unstable and network activity cannot be adjusted anymore by an external offset current. A possible solution here is to employ a stochastic neuron model that allows control of its firing statistics.

Performance model

The ability to describe the performance characteristics of the HNC node and predict its behavior for different parameter sets and under varying workload conditions is highly valuable. The model has proven to be highly accurate, with its formulation made possible by the RTL-level design methodology, which exposes detailed insights into the latencies of the HNC node microarchitecture.

Accompanied by performance measurements, the performance model allowed a verification of the implementation in terms of expected performance, as well as it quantifies the performance-determining processes. By extending the model to include the additional communication and synchronization latencies occurring in a multi-node system, conclusions could also be drawn for cluster operation. To this end, several simplifying assumptions were made, in particular, simplifications with regard to inter-node communication latency values. The value of 500 ns assumed for the transmission latency time is ambitious – network technologies are typically optimized for throughput, but not for latency. However, these choices are sufficient to demonstrate the effect of inter-node communication as a performance-determining factor, as illustrated in Figure 5.3 in the respective contributions of the related processes to the loss of performance. Irrespective of this, the model also reveals that external memory access is the dominant performance determining factor.

For the selected technology – the AMD Xilinx Zynq-7000 SoC device family – the performance model estimates an acceleration factor on the order of 10 to 50 for medium and small workloads. Such a workload is created, for example, by a network consisting of $N = 10{,}000$ neurons, where the neurons have an average firing rate of $\bar{\nu} \approx 2 \ldots 10$ spks/s. To simulate such a network, 10 HNC nodes would need to be clustered. The performance model here also reveals that the achievable acceleration and the reasonable size of a cluster are limited by the external memory bandwidth.

Model Parameter	NetFPGA Value	Comment
f_{clk}	189.383 MHz	clock frequency
h	0.1 ms	simulation resolution
$\bar{\nu}_k$	≈ 25 spks/h	workload generated by the microcircuit model (see Section 6.3.2)
L_{Σ}^{SE}	≈ 285 clock cycles	ODE solver latency ($\approx 1.5\,\mu$s, neurons per worker 255, pipeline stage count 30)
L_{S}	≈ 8 clock cycles	data read latency retrieving a single list of synaptic targets; the value is estimated as $L_{\text{S}} = (f_{\text{clk}} K_{\text{total}} w_{\text{len}})/(BMN)$ based on: • the achieved memory bandwidth, $B \approx 20$ GB/s (taken from Kauth et al. (2023), Figure 8) • the word length of a synaptic data item, $w_{\text{len}} = 56$ bit • the number of NetFPGA nodes, $M = 35$ • the number of neurons simulated, $N = 77,169$, and • the number of synapses, $K_{\text{total}} \approx 299 \cdot 10^6$ (see also Section 6.3.2)
L_{COM}	≈ 406 clock cycles	communication latency ($\approx 2.144\,\mu$s, synchronization time)
α	0	not given

Table 5.2 | NetFPGA latency values and corresponding performance model parameters. Hardware parameters and measured latencies were derived from the descriptions given in Kauth et al. (2023).

Although the formulation of the performance model is specific to the HNC node, it captures in a generic way the performance-determining aspects of discrete-time neural network simulations of point neuron models. Thus, the idea can be adapted and applied to other *simulation engines* of this kind, as the following example may show.

In Kauth et al. (2023), a simulation of the cortical microcircuit model implemented on a cluster of 35 NetFPGA SUME boards is presented. Based on the simulation engine's hardware properties and the workload that is generated by the simulated model, the performance model can provide a good estimate of the achievable acceleration factor. Table 5.2 summarizes the relevant latencies of the NetFPGA cluster implementation derived from the descriptions given in Kauth et al. (2023). Also shown are the corresponding performance model parameters. From these parameters and using Equation (5.9) we derive

$$F_{\text{C}}(\bar{\nu}_{\text{k}}) = \frac{h f_{clk}}{L_{\Sigma}^{\text{SE}} + \bar{\nu}_{\text{k}}(L_{\text{S}} + \alpha L_{\text{COM}}) + L_{\text{COM}}} = \frac{0.1\ \text{ms} \cdot 189.383\ \text{MHz}}{285 + 25 \cdot 8 + 406} \approx 21. \tag{5.14}$$

This is in agreement with the acceleration factor of about 20 that the authors have measured.

Guiding design decisions – options for performance improvements

By offloading performance-critical tasks to programmable logic, the HNC node design aims to remove limitations that are a consequence of the von Neumann bottleneck in conventional computer architectures. The close proximity of data and operations, hardware-level parallelization, and pipelining are the ingredients from which the HNC node derives its performance. This is undermined to some extend by storing the network connectivity data in an external memory – a design decision that was based on technical constraints and application requirements (see Section 3.3.1). It revealed to largely determine performance. Toward higher workloads, performance becomes bound by the access latency and limited bandwidth of this external memory.

The evaluations conducted suggest a number of improvements and approaches that can mitigate the negative effects:

Reducing data sizes: All numbers are stored in a 40-bit fixed-point data format to provide sufficient numerical accuracy. This decision can certainly be questioned. While the state variables and constants of the Izhikevich neuron model require this precision (see Section 2.3.4.2), the values of synaptic weights may not and can probably be stored using a less precise format. There is evidence that 32-bit fixed point is already sufficient – the result achieved at the end of Iteration III of the worked example (see Section 2.3.4.3) can be interpreted in this manner. This would reduce the size of an LST element by 12.5% and correspondingly reduce the required external memory bandwidth. Depending on the neuron and network model, a further reduction in precision may even be possible (see, e.g., Dasbach et al., 2021). There is still a debate in the neuromorphic community about how numerical precision affects neuron model dynamics and the results of neural network simulations, and what numerical precision is required. A conclusive answer to these questions has not yet been given.

Data compression is also an option and could be done conveniently in hardware. Applied to all LSTs, it can significantly reduce the amount of data read from external memory. From the performance model, we can derive that a 50% reduction ($L_S \rightarrow L_S/2$), for example, will increase performance by 80% for high workload. According to Equation (5.9) and the *prototype* parameter set (see Table 5.1), for cluster operation and depending on workload, the following acceleration factors can be expected: low workload, $F_C(\bar{\nu}_k = 1) = 83$ (+23%); medium workload, $F_C(10) = 25.6$ (+70%); high workload, $F_C(20) = 14.5$ (+80%).

Hiding latencies: The HNC node design does not exploit that the processing of synaptic events with $D > D_{min}$ can be delayed, and only events with $D = D_{min}$ need to be processed in the current simulation time step. This provides an option to hide memory access latency behind communication latency. A performance improvement would be seen at medium workloads. For

high workloads, the effect would be minimal because memory access latency dominates.

Exploiting advances in device architecture and technology: The performance of the HNC node is bound by memory latency. Migrating the design to a more powerful SoC architecture that provides higher memory bandwidth is therefore an attractive option. There is a variety of device families and options available. Here are two examples: the AMD Xilinx Zynq UltraScale+ device family (AMD Xilinx, 2023c); and the AMD Xilinx Versal HBM adaptive SoC series (AMD Xilinx, 2023a).

The AMD Xilinx Zynq UltraScale+ device family builds on the Zynq-7000 architecture, which is an advantage for migrating the design. The devices support high-performance external memories. The memory bandwidth for this technology is specified as 19.2 GB/s (AMD Xilinx, 2023d), which is more than eight times the bandwidth that the HNC node prototype practically could utilize (see Section 3.3.4.2). The UltraScale+ device architecture also comprises a 64-bit ARM Cortex-A53 quad-core application processing unit with an advanced SIMD and floating-point extension. The hardware and software mixed HNC node architecture can benefit from these features, e.g., for the implementation of plasticity algorithms.

As a recent development in FPGA-SoC technology, the AMD Xilinx Versal HBM adaptive SoC series integrates high-bandwidth memory (HBM) technology that allows up to 819 GB/s of memory bandwidth (AMD Xilinx, 2023b). These devices also provide a large amount of programmable logic resources and a powerful ARM-based processing system (AMD Xilinx, 2024).

Migrating and adapting the HNC node design to more advanced SoC technology is certainly the most attractive of the options outlined above. In particular, the integration of HBM technology holds great potential and promises to enable architectures capable of substantially accelerating simulations, even on a large scale and high workloads (see Chaper 6).

Chapter 6

Toward Large Scale

6.1 Introduction

The HNC node architecture provides flexibility in the algorithmic implementation of models, is open to extensions, and is capable of delivering significant acceleration – key requirements for a neuromorphic platform dedicated to neuroscience simulation. Equally important is scalability. Here, the Zynq-7000 SoC device chosen to implement the HNC node prototype imposes a limit on the number of neurons per node as well as the achievable acceleration. Both are bound to memory, limited by memory size (on-chip) and memory bandwidth (off-chip). This restricts the size of a network for which significant acceleration can be achieved. Acceleration factors of 10 to 50 can be realized in simulations of networks with several tens of thousands of neurons, but hardly more (see Chapter 5). Advances in FPGA-SoC technology, the integration of high-bandwidth memory (HBM) and the availability of large on-chip memories promise to overcome these limitations, enabling larger system sizes and allowing for higher acceleration factors.

Scaling up system size for the simulation of large-scale networks – networks with hundreds of thousands or even millions of neurons – requires a higher neuron density per node to keep the system size manageable. This increases the amount of memory needed on a node. In this regard, the use of true dual-port block RAM in the HNC node design needs to be reconsidered, as it is typically not available in the required size. In addition, larger networks and a higher number of neurons per node both increase the computational load on a node. To cope with these higher workloads, further parallelization strategies need to be devised to keep latencies low.

As an extension to the HNC node design, additional parallelization options are presented in this chapter. Alternative ring buffer memory architectures that do not rely on true dual-port memory but achieve similar throughput are discussed. For the proposed architectural enhancements, estimates of the achievable performance are presented. These estimates are based on a workload analysis of two large neural network models used in neuroscience.

At the intended scale, system integration also becomes an important aspect that affects overall system performance. For example, large amounts of data need to be moved efficiently; into the system in the form of connectivity data, and out of the system in the form of simulation data. Therefore, aspects of system integration are finally discussed and a basic architecture concept for an integration of a neuromorphic computing (NC) system into the high-performance computing (HPC) landscape is presented.

Contributions

- Additional parallelization options and alternative ring buffer memory architectures are presented that enable designs capable of coping with the high workloads generated by

large-scale networks.

- Presented is a workload analysis of two neural network models widely used in neuroscience, the *microcircuit model* (Potjans and Diesmann, 2014), and the *multi-area model* Schmidt et al. (2018a,b); Schuecker et al. (2017).
- Based on this workload analysis, performance estimates for the proposed architectural enhancements are provided, and it is shown that recent commercial off-the-shelf FPGA-SoC technology is a suitable substrate capable of delivering significant acceleration even for large-scale workloads.
- A basic architectural concept for an integration of an NC system into the HPC landscape is proposed.

6.2 Architectural Enhancements

6.2.1 Further Parallelization Options

The most compute-intensive part of a simulation is the processing of incoming spike events. The computational load that a large network with natural dense connectivity here generates is on the order of several thousand synapse updates per simulation time step and compute node. The HNC node parallelizes these computations and distributes the workload over multiple processing units. Within a processing unit, excitatory and inhibitory synaptic events are processed sequentially as they are passed through the ring buffer (RB) pipeline. To achieve significant acceleration, the latency of this processing must be as low as possible. This latency can be reduced by further increasing the degree of parallelism. An option to achieve this is a parallelization of the processing at the level of spike events. We can process excitatory and inhibitory events simultaneously, exploiting that a neuron forms either excitatory or inhibitory synaptic connections at all of its axonal branches – a functional consequence of Dale's principle[1]. Note that the RB architecture already separates a neuron's excitatory and inhibitory synaptic inputs (see Section 3.3.4.1).

Figure 6.1A illustrates the idea. The lists of synaptic targets (LSTs) in memory are divided into two groups, LSTs of excitatory presynaptic neurons and LSTs of inhibitory presynaptic neurons. Within a processing unit, each of the two LST types is assigned its own RB pipeline and RB memory to enable their simultaneous processing. The structure formed by an RB pipeline and an RB memory is hereafter referred to as an RB block (RBB), where a processing unit then contains two RBBs that form an RB unit (RBU) (indicated by the dashed frames in Figure 6.1A). Although the number of excitatory connections dominates the number of inhibitory connections in neural

[1] The principle states that a neuron performs identical chemical actions at all of its synaptic connections to other cells.

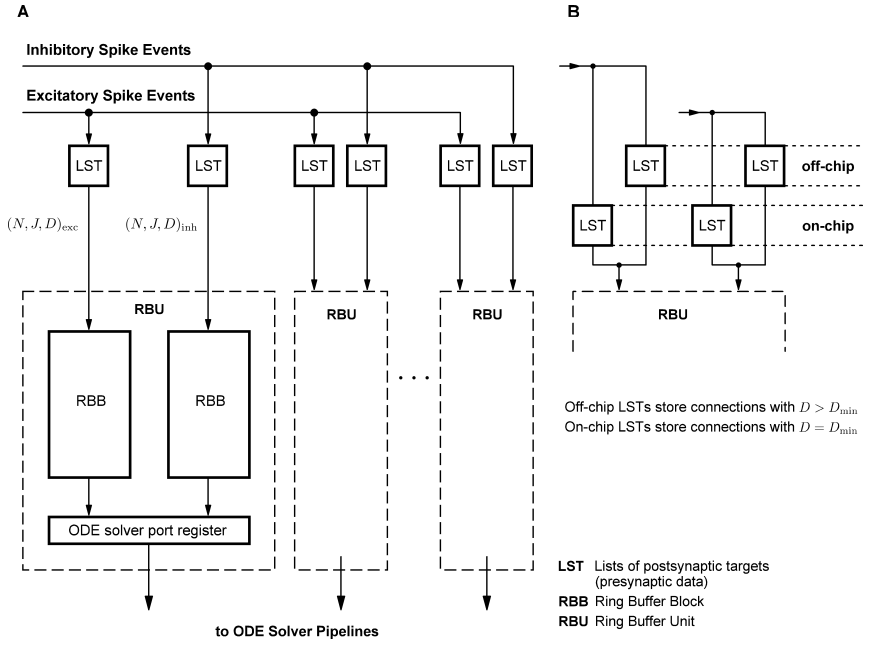


Figure 6.1 | Parallelization of the processing of incoming spike events. To further parallelize spike processing, two architectural enhancements are proposed: (**A**) excitatory and inhibitory spike events are processed simultaneously exploiting that a neuron forms either excitatory or inhibitory synaptic connections at all of its axonal branches; and (**B**) presynaptic data items (stored in the LSTs) that need to be processed in the current simulation time step, i.e., synaptic events with $D = D_{\text{min}}$, are held on-chip in fast memory to allow their immediate processing while reading the data items with $D > D_{\text{min}}$ from off-chip external memory in parallel. In this way, external memory access latency can partially be hidden.

network models, the workload for a processing unit's RBBs is not necessarily unbalanced because inhibitory neurons usually have higher firing rates. The size of the RB memory of a processing unit remains the same, but it is now divided between the two RBBs within an RBU.

This architectural enhancement requires support from the spike communication fabric, which must provide excitatory and inhibitory spike events in parallel and indicate the type of a spike event. To be effective, the additional parallelism also requires a higher memory bandwidth.

Not only is memory bandwidth an issue, arbitration and scheduling of memory requests introduces an access latency. This memory access latency can be mitigated by taking advantage of the fact that only the presynaptic data items of connections with $D = D_{\text{min}}$ must be processed in the current simulation time step; all others can be delayed. If these data items are held on-chip in

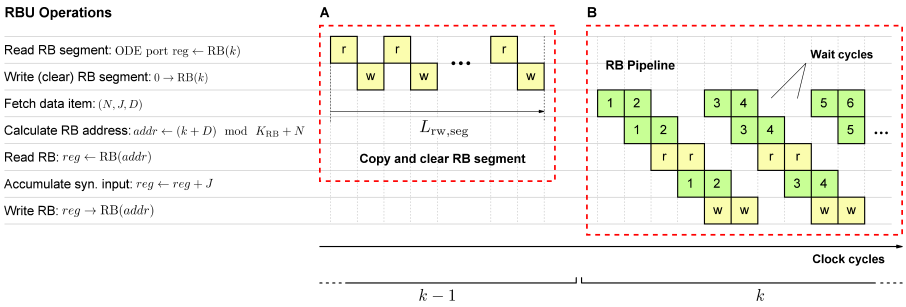


Figure 6.2 | Timing diagram of RB update sequences. Without true dual-port capability, RB read and write operations (marked yellow) serializes. Shown are two sequences of RBU operations: (**A**) a sequence of reads and writes at the end of a simulation time step (here, the $(k-1)^{\text{th}}$ time step) provides the ODE solver(s) with the weighted synaptic inputs and clears the entire segment for reuse in the next simulation time step; and (**B**) a possible sequence of RB pipeline operations processing the presynaptic data and updating RB entries. Due to the serialization of reads and writes, the RB pipeline has to include *wait cycles*. In sequence (**A**), $L_{\text{rw,seg}}$ denotes the latency introduced by the operations in number of clock cycles.

fast memories, external memory access latencies can be hidden. Connections with $D = D_{\text{min}}$ can then be processed without delay, while connections with $D > D_{\text{min}}$ can be read in parallel, buffered and processed later. This divides an LST, in an on-chip and an off-chip part. Figure 6.1B illustrates this.

The additional on-chip memory required remains moderate. For instance, in the cortical microcircuit model (Potjans and Diesmann, 2014), 1.33%[2] of the connections have a delay of $D = D_{\text{min}}$. The model is biologically plausible and exhibits naturally dense connectivity. Given this, a compute node capable of processing 4000 neurons – each forming 10,000 connections – would require approximately 4.3 MB of additional on-chip memory, assuming a 64-bit word length for an LST element.

6.2.2 Memory Partitioning

The HNC node design leverages the true dual-port feature of BRAMs to achieve the best possible RB pipeline throughput, i.e., a pipeline initiation interval equal to 1. The use of true dual-port memory is a problematic choice when aiming for a large-scale system where neuron density per node needs to be increased to keep system size manageable and communication latency low. Increasing the number of neurons per node results in the need for larger amount of BRAM, which is a limited resource in an FPGA. True dual-port BRAMs are built from dual-port SRAM cells,

[2]This value was derived by extracting the connectivity from a NEST simulation of the model.

which require more transistors per bit and thus more chip area. Therefore, their memory sizes are smaller and the number of blocks is limited. However, similar RB pipeline throughput can also be achieved with RB memories that are implemented as single-port.

A single-port memory allows only one write or read operation at a time. This requires *wait cycles* to be included in the pipeline operation when accessing RB memory. Thus, we need to reconsider the sequence(s) of operations performed by both the RB pipeline and the ODE solver pipeline. We will base this on the RB architecture variant presented in Section 3.3.4.3, Figure 3.14B (delete an entire RB segment before it is reused).

Figure 6.2 shows the timing diagram of two sequences of access patterns for a serialized RB memory access in one possible order. In each clock cycle, a memory read or write operation is performed (indicated by the yellow boxes). A number of clock cycles $L_{\text{rw,seg}}$ is required to transfer the content of an entire RB segment to the ODE solver pipeline(s) and to clear the segment for reuse. The sequence is shown in Figure 6.2A. The operations are listed on the left side of the figure. Figure 6.2B shows a possible sequence of operations that an RB pipeline performs when processing presynaptic data items (items are numbered 1 through 6).

It is obvious that due to the serialization of reads and writes the initiation interval of the RB pipeline is 2 – regardless of the order of the operations. This halves the pipeline throughput and that of an RBB, respectively. In addition, sequentially reading and writing an entire RB segment adds the latency $L_{\text{rw,seg}}$. Both pipeline throughput and sequential read/write access latency can be optimized by using memory partitioning and by considering access patterns in the memory configurations.

Architecture alternatives

We can reduce the initiation interval of an RB pipeline or RBB, respectively, by partitioning the RB memory. A sensible approach here is to subdivide the groups of neurons associated with an RBU and assign each group its own memory partition, i.e., we subdivide RB segments.

The following discusses four architecture alternatives with different arrangements of pipelines and memory partitions: (i) single memory; (ii) partitioned memory with interleaved access; (iii) partitioned memory with parallel access and (two) parallel pipelines; and (iv) partitioned memory with parallel access and a single pipeline. The four architecture variants are shown in Figure 6.3.

They vary in the following aspects:

Single Memory: This architecture variant is the non-partitioned case with a single memory block assigned to a single pipeline. It serves here as reference. Pipeline operation corresponds to the sequence of operations outlined in Figure 6.2B for which an initiation interval of II = 2 applies.

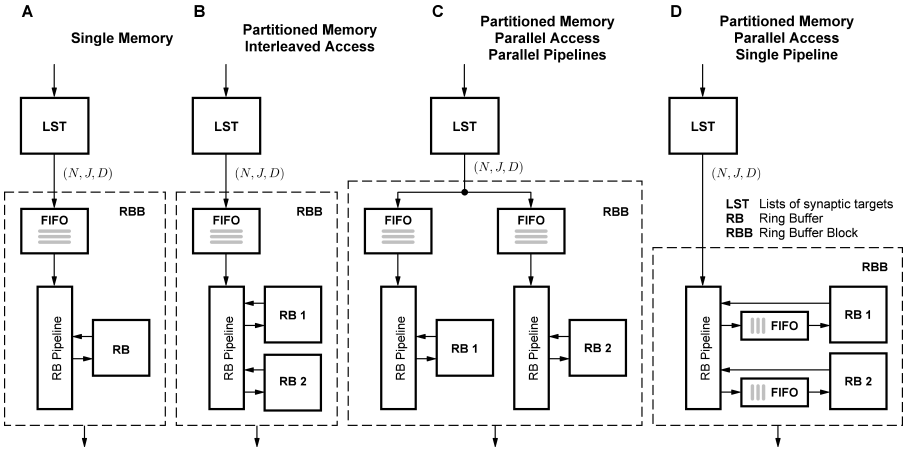


Figure 6.3 | Partitioning of RB memory. Alternative ring buffer block (RBB) architectures and assignment of RB memory to RB pipelines: (**A**) non-partitioned case with a single memory block assigned to a single pipeline; (**B**) single pipeline, where memory is divided into two memory partitions (RB 1 and RB 2) that are accessed in an interleaved fashion; (**C**) two pipelines, each assigned to one memory partition for parallel, independent access; and (**D**) similar to (**C**), but moves the FIFO buffers to the write paths of the partitions, saving a pipeline. See also the main text for description.

Partitioned Memory with Interleaved Access: RB memory is divided into two partitions. Both share a processing pipeline. Partitioning is done such that segments can be accessed in an interleaved fashion. Presynaptic data items are buffered in a FIFO-lane. The next item in a sequence is fetched if the corresponding memory partition is not blocked. There is a 50% probability that this is the case. We can thus expect a 25% reduction in latency in comparison with the *single memory* variant, i.e., we can expect an initiation interval of II = 1.5.

Partitioned Memory with Parallel Access and (two) Parallel Pipelines: RB memory is divided into two partitions with each assigned its own processing pipeline. This architecture alternative allows parallel and independent access to two RB segments.

Partitioned Memory with Parallel Access and Single Pipeline: Parallel access to segments can also be achieved with a single pipeline by moving the FIFO buffers into the write paths of the memory partitions. The working principle is the following. Read operations have priority over write operations and can always be executed without delay. A buffering of presynaptic data items on entry of the processing pipeline is therefore not necessary. Write operations are queued in FIFO-lanes and executed when the memory partition they are connected to is not blocked by a read operation. A disadvantage of this solution is the increased likelihood of potential

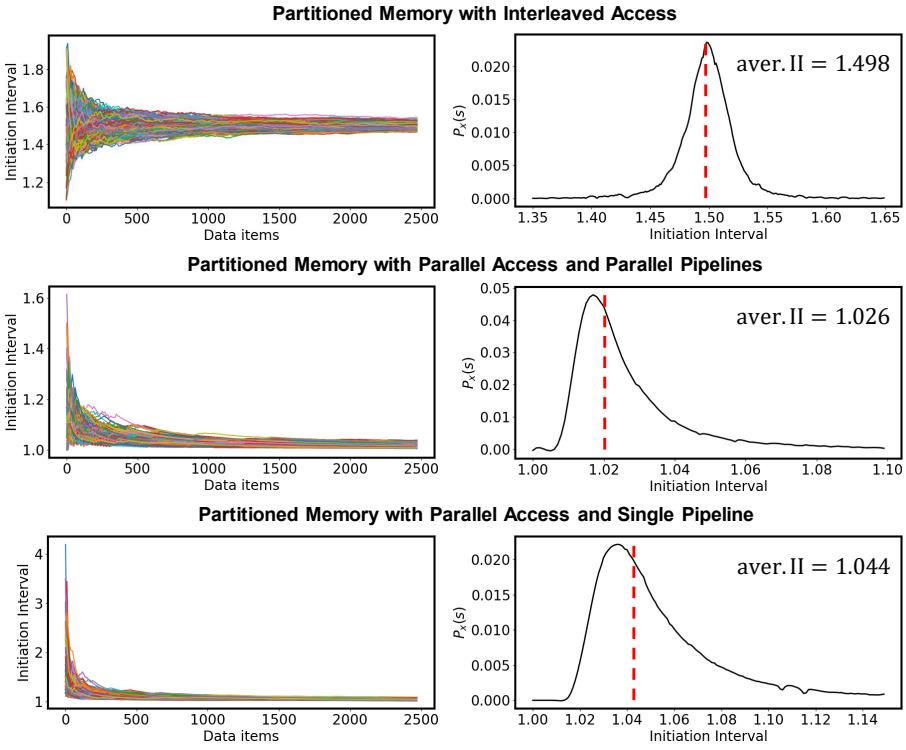


Figure 6.4 | Monte Carlo simulations. The initiation intervals II of the three architectures that use memory partitioning have been determined by subjecting software models of the architecture variants to Monte Carlo simulations. For each architecture variant, II was determined from 1500 Monte Carlo trials. In each Monte Carlo trial, 2500 data items were passed through the RBB pipeline(s) and evenly distributed over memory partitions. The left panels show the plots of the Monte Carlo trials performed for each of the three architecture variants. The right panels display the corresponding probability distributions of II, where the red dashed lines mark their calculated average values.

read-before-write conflicts because write operations are delayed.

Monte Carlo simulations

Except for the *Single Memory* variant (Figure 6.3A), the initiation intervals are not constant and depend on the distribution of data items across memory partitions. In order to determine the behavior of the different architectures, their processing logic was implemented in software. The initiation intervals were then derived from Monte Carlo experiments. The results of the Monte Carlo simulations are shown in Figure 6.4.

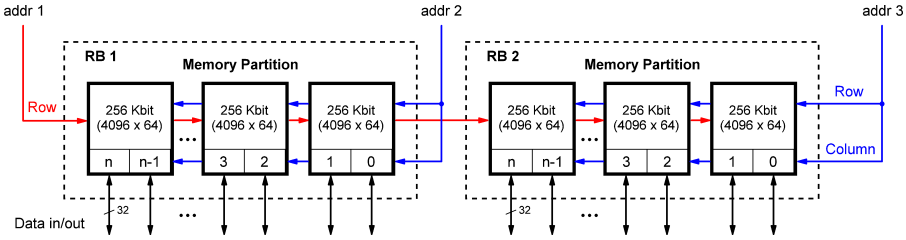


Figure 6.5 | Partitioned RB memory addressing. Memory blocks are concatenated across memory partitions (RB1 and RB2). This allows two addressing schemes: a complete memory row can be addressed (indicated by the red arrows, addr 1) to read and write an entire RB segment, or a larger part of it, at once; and two RB entries, one in each partition, can be addressed simultaneously (indicated by the blue arrows, addr 2 and addr 3).

The *Interleaved Partitioned Memory* variant (Figure 6.3B) achieves an initiation interval close to II = 1.5 – as we had expected. The architecture alternatives that implement parallel access (Figure 6.3C and D) both reach an average initiation interval close to the ideal value, i.e., II = 1. Further partitioning of the memory is therefore not useful. The architectures that implement parallel access differ only in the resources required, where the advantage of having only one pipeline is opposed by the need for larger FIFO buffer sizes. The FIFO buffer utilization – the maximum required size – was also estimated by the Monte Carle simulations. The complete data set can be found in Appendix **E**.

Memory configurations

From the two sequences displayed in Figure 6.2, two different memory access patterns can be observed: a regular, sequential access pattern (Figure 6.2A); and an irregular access pattern (Figure 6.2B). The first introduces the latency $L_{\text{rw,seg}}$. This latency is significant. Sequentially reading and writing all entries of an RB segment requires $2N^{\text{RBU}}$ clock cycles, i.e., two times the number of neurons associated with an RBU. The second sequence updates RB entries on a stochastic basis across RB segments. Here, latency depends on workload.

Latency can be reduced by considering these memory access patterns in the architecture and configuration of the RB memories. Figure 6.5 shows a possible arrangement of memory blocks and partitions that takes this into account; here, using 256 Kbit memory blocks in a 4096 bit x 64 bit organization (similar sizes are provided by FPGAs)[3]. The setup allows for two addressing schemes: (i) the addressing of an entire RB segment, or a larger part of it, at once;

[3]The AMD Xilinx UltraScale architecture provides memory primitives in a 4096 bit x 72 bit configuration, for example (AMD Xilinx, 2021a).

Parameter	Value
Number of neurons per node, N^{M}	4096
Minimum synaptic transmission delay, D_{min}	0.1 ms
Maximum synaptic transmission delay, D_{max}	51.1 ms
Simulation resolution, and resolution of delays, h	0.1 ms
Word length of synaptic inputs, $w_{\mathrm{len,J}}$	32 bit
Organization of memory blocks	4096 bit x 64 bit
Word length of a synaptic target list element, $w_{\mathrm{len,LST}}$	64 bit
Clock frequency, f_{clk}	200 MHz

Table 6.1 | Defined set of node parameters.

and (ii) the addressing of two individual RB entries in the two partitions simultaneously. The first serves the sequential access pattern. Memory blocks are concatenated across partitions maximizing the accessible word length. Addressing a complete row (in Figure 6.5 indicated by the red arrows, addr 1) avoids having to read the entries of an entire RB segment sequentially, hence reducing the latency $L_{\mathrm{rw,seg}}$. The second serves the irregular access pattern of RB updates. In each partition, an RB entry can be addressed simultaneously by its row and column address (in Figure 6.5 indicated by the blue arrows, addr 2 and addr 3).

In order to demonstrate the effect of different memory configurations on the latency $L_{\mathrm{rw,seg}}$, we define a set of node parameters. The parameter set is shown in Table 6.1 and represents a realistic specification with regard to the design space. From this specification, a node's total amount of RB memory can be calculated by

$$S_{\mathrm{RB,total}} = N^{\mathrm{M}} K_{\mathrm{RB}} 2 w_{\mathrm{len,J}}, \quad \text{with} \tag{6.1}$$

$$K_{\mathrm{RB}} = \frac{D_{\mathrm{max}}}{h} + 1, \tag{6.2}$$

where K_{RB} is the number of RB segments, which is determined by the maximum supported synaptic transmission delay D_{max} and the resolution h of the delay values. In Equation (6.1), N^{M} denotes the number of neurons per node, and $w_{\mathrm{len,J}}$ is the word length of synaptic inputs (excitatory and inhibitory). For the defined parameter set, the calculation yields $S_{\mathrm{RB,total}} = 128$ Mbit.

This memory is distributed among the RBUs. Note, that an RBU consists of two RBBs (see Figure 6.1), where an RBB comprises two partitions of chained memory blocks. Depending on the number of RBUs, this results in different memory configurations and different accessible RB memory word lengths. Table 6.2 lists the resulting configurations, values of $L_{\mathrm{rw,seg}}$, and clock

RBUs	Neurons per RBU	Memory per RBU	Memory Blocks per Partition	Organization of Partition	$L_{\text{rw,seg}}$ (clock cycles)	Clock Cycles Saved
8	512	16 Mbit	16	4096 bit x 1024 bit	16	1008
16	256	8 Mbit	8	4096 bit x 512 bit	16	496
32	128	4 Mbit	4	4096 bit x 256 bit	16	240
64	64	2 Mbit	2	4096 bit x 128 bit	16	112
128	32	1 Mbit	1	4096 bit x 64 bit	16	48

Table 6.2 | RB memory configurations. RB memory configurations for different numbers of RBUs and clock cycles saved per simulation time step for the given set of node parameters listed in Table 6.1.

cycles saved for different numbers of RBUs.

The number of clock cycles to read and write an entire RB segment here results in $L_{\text{rw,seg}} = 16$ and is the same for all configurations – the number of concatenated memory blocks per partition grows with the number of neurons assigned to an RBU, keeping $L_{\text{rw,seg}}$ constant. The number of clock cycles that can be saved compared to sequential read and write is two times the number of neurons assigned to an RBU minus 16. The value decreases with the number of RBUs, but is still considerably 48 clock cycles for a configuration that comprises 128 RBUs.

6.3 Performance Estimation

To examine the effectiveness of the proposed architectural enhancements and their ability to deliver substantial acceleration under the workloads generated by large-scale networks, it is necessary to have a definition of these workloads. This includes defining an appropriate workload measure and range of values that represents the workloads.

6.3.1 Measure of Workload

At the intended scale, the metric introduced by the workload model in Section 5.2.1 cannot be applied. The average number of postsynaptic targets per presynaptic neuron on a node is no longer independent of network size, nor can we assume a uniform distribution of connections. A measure of workload must therefore take connectivity into account. Instead of the number of spike events per simulation time step, we here define as a measure of a node's workload the average number of presynaptic data items (synaptic events) that a node has to process in a simulation time step, denoted $\bar{U}^{\text{M}}$ in the following.

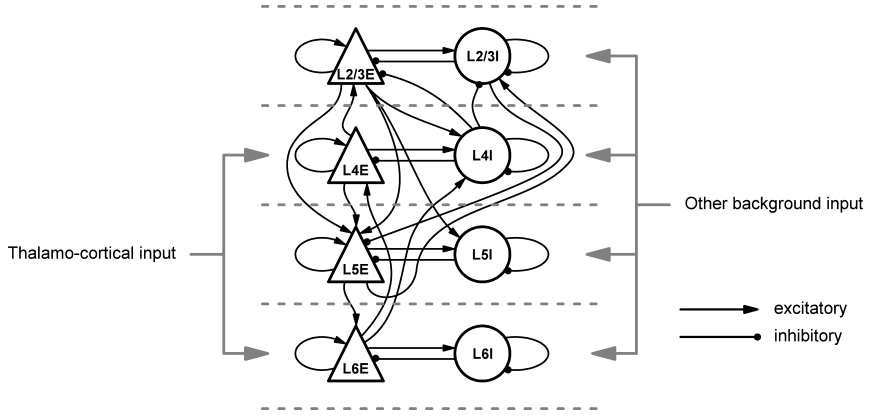


Figure 6.6 | Microcircuit model. The model is organized into four layers (L2/3, L4, L5, and L6), each consisting of two populations, an excitatory (E) and an inhibitory (I) population. The model comprises 77,169 LIF neurons and approximately 299 million synapses. For connections, only those with a probability of at least 4% are shown, where pointed arrowheads indicate excitatory connections, and endpoints represent inhibitory connections. The model receives background input from a stationary Poisson point process with constant rate.

6.3.2 Workloads of Large-Scale Networks

A value or value range that reflects representative workloads can be estimated by studying networks of the relevant scale. To this end, two neural network models widely used in neuroscience are considered. A model of early sensory cortex published by Potjans and Diesmann (2014), also known as the *microcircuit model*, is examined in this regard. The results are then extrapolated to a model of the macaque visual cortex described in Schmidt et al. (2018a,b); Schuecker et al. (2017), hereafter referred to as *multi-area model*.

Microcircuit model

The microcircuit model represents the cortical network beneath a 1 mm^2 surface of early mammalian sensory cortex and features full natural dense connectivity. The model comprises 77,169 LIF neurons and about 299 million synapses, and is organized into four layers (L2/3, L4, L5, and L6). Each layer consists of two populations, an excitatory (E) and an inhibitory (I) population, resulting in eight distinct populations. A sketch of the model is shown in Figure 6.6. Synaptic connections between populations are defined by connection probabilities. Synaptic strengths and propagation delays, which are drawn from uniform distributions, are not relevant for estimating workload and are therefore not considered further.

Population	L2/3E	L2/3I	L4E	L4I	L5E	L5I	L6E	L6I
Population size, N_X	20 683	5834	21 915	5479	4850	1065	14 395	2948
Average firing rate, $\bar{\nu}_X$	0.903	2.965	4.414	5.876	7.596	8.633	1.105	7.829

Table 6.3 | Microcircuit population sizes and average firing rates. Population sizes according to Potjans and Diesmann (2014), and population-specific average firing rates obtained from a NEST simulation using the implementation of van Albada et al. (2018).

The workload that a simulation of the microcircuit model places on a compute node can be estimated by examining the model's connectivity and firing statistics. I present two approaches here to determining this workload: (i) a calculation that takes into account the connectivity between populations and the population-specific firing statistics; and (ii) a simplified calculation that assumes uniform connectivity across populations and uses the average firing rate over all neurons in the network. I demonstrate that, for the microcircuit model, both approaches yield the same result.

Workload estimation considering the connectivity between populations

For a workload estimate, the following network properties need to be known: the population sizes; the average firing rate of the neurons in each population; and the number of synapses between populations. The population sizes are given in Table 6.3 according to Potjans and Diesmann (2014). Also shown are the population-specific average firing rates of neurons, which have been obtained from a NEST simulation using the implementation of van Albada et al. (2018). The absolute number of synapses between a pair of populations K_{YX} can be calculated from their connection probability C_{YX}. Connection probability and number of synapses are related as (see Potjans and Diesmann, 2014)

$$C_{YX} = 1 - \left(1 - \frac{1}{N_X N_Y}\right)^{K_{YX}}, \tag{6.3}$$

where N_X and N_Y denote the number of neurons of the presynaptic (X) and postsynaptic (Y) population with $\{X, Y\} \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E,I}\}$.

The absolute number of synapses between a pair of populations is then given by

$$K_{YX} = \frac{\ln(1 - C_{YX})}{\ln(1 - \frac{1}{N_X N_Y})}. \tag{6.4}$$

Table 6.4 shows the complete connectivity map of the microcircuit with the connection probabilities and the resulting absolute synapse counts.

		From *X*							
		L2/3E	L2/3I	L4E	L4I	L5E	L5I	L6E	L6I
To *Y*	L2/3E	0.1009 45 499 804	0.1689 22 323 576	0.0437 20 253 647	0.0818 9 670 918	0.0323 3 293 577	0.0 0	0.0076 2 271 403	0.0 0
	L2/3I	0.1346 17 443 694	0.1371 5 018 762	0.0316 4 105 338	0.0515 1 690 073	0.0755 2 221 212	0.0 0	0.0042 353 460	0.0 0
	L4E	0.0077 3 503 669	0.0059 756 561	0.0497 24 482 849	0.135 17 413 575	0.0067 714 524	0.0003 7002	0.0453 14 624 431	0.0 0
	L4I	0.0691 8 114 253	0.0029 92 831	0.0794 9 933 537	0.1597 5 223 271	0.0033 87 836	0.0 0	0.1057 8 810 905	0.0 0
	L5E	0.1004 10 613 575	0.0622 1 817 058	0.0505 5 507 804	0.0057 151 900	0.0831 2 040 738	0.3726 2 407 889	0.0204 1 438 969	0.0 0
	L5I	0.0548 1 241 436	0.0269 169 424	0.0257 607 666	0.0022 12 851	0.06 319 601	0.3158 430 443	0.0086 132 414	0.0 0
	L6E	0.0156 4 681 225	0.0066 556 108	0.0211 6 727 569	0.0166 1 320 233	0.0572 4 112 224	0.0197 305 028	0.0396 8 372 649	0.2252 10 827 677
	L6I	0.0364 2 260 836	0.001 17 207	0.0034 220 032	0.0005 8078	0.0277 401 637	0.008 25 217	0.0658 2 888 426	0.1443 1 354 319

Table 6.4 | Microcircuit connectivity map. Connection probabilities C_{YX} between presynaptic X and postsynaptic Y populations (upper rows) according to Potjans and Diesmann (2014), and resulting absolute number of synapses K_{YX} between any pair of populations (lower rows).

If the average firing rates within populations are known, we can calculate the average number of synapse updates per simulation time step for each of the populations, i.e., the workload *arriving* at a population $Y \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E,I}\}$. We will divide the workload of a population into an excitatory and an inhibitory component here, as this separation is needed later in the performance estimation of the architecture variants. The excitatory workload arriving at a population can be formulated as

$$\bar{U}^{\text{POP}}_{\text{exc},Y} = h \sum_X \bar{\nu}_X K_{YX} \quad \forall\, X \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E}\}, \\ \forall\, Y \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E,I}\}, \tag{6.5}$$

and the inhibitory workload accordingly as

$$\bar{U}^{\text{POP}}_{\text{inh},Y} = h \sum_X \bar{\nu}_X K_{YX} \quad \forall\, X \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{I}\}, \\ \forall\, Y \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E,I}\}, \tag{6.6}$$

where $\bar{\nu}_X$ is the average firing rate of the neurons in population X, K_{YX} is the number of synapses from population X to Y, and h specifies the resolution of the simulation. The total workload that

Population	L2/3E	L2/3I	L4E	L4I	L5E	L5I	L6E	L6I
$\bar{U}^{\mathrm{POP}}_{\mathrm{exc},Y}$	15 798	5113	13 279	6155	5098	636	7439	925
$\bar{U}^{\mathrm{POP}}_{\mathrm{in},Y}$	12 300	2481	10 462	3096	2705	428	9678	1090
$\bar{U}^{\mathrm{POP}}_{Y}$	28 098	7594	23 741	9251	7803	1064	17 117	2015

Table 6.5 | Microcircuit population workload. Average number of synapse updates per simulation time step specific to each population.

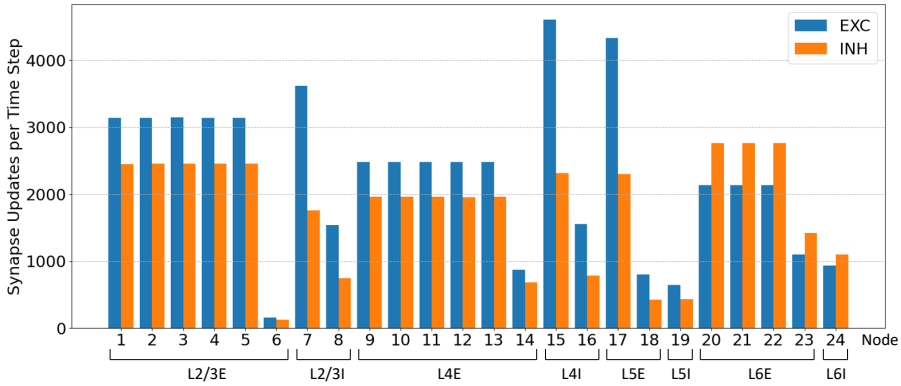


Figure 6.7 | Distribution of workload. Calculated distribution of workload on a cluster of 24 nodes when simulating the cortical microcircuit model, where each node is capable of processing 4096 neurons. Populations are assigned to distinct nodes. The data has been acquired twice: from a NEST simulation (single node), here derived from the exported connectivity data and the spike recordings of 60 second simulated time; and calculated based on the Equations (6.5) and (6.6). Both resulted in nearly identical charts. The one shown is based on the NEST simulation.

is arriving at each population is then given by

$$\bar{U}^{\mathrm{POP}}_{Y} = \bar{U}^{\mathrm{POP}}_{\mathrm{exc},Y} + \bar{U}^{\mathrm{POP}}_{\mathrm{inh},Y} \quad \forall\, Y \in \{\mathrm{L2/3, L4, L5, L6}\} \times \{\mathrm{E,I}\}. \tag{6.7}$$

The calculated values are summarized in Table 6.5. These values show that the workload generated by the microcircuit is not uniform across populations, nor is there a perfect balance between excitatory and inhibitory events. The chart diagram in Figure 6.7 illustrates this. It shows how the workload would be distributed in a cluster of 24 nodes, where each node is capable of simulating 4096 neurons, and nodes are assigned to distinct populations. In this setup, node 15, which is assigned to population L4I, has a much higher workload to process than any other node and will therefore determine the speed of the simulation.

By shuffling populations and assigning neurons to nodes in a round-robin fashion, for example,

workload can be balanced and simulation runtime can be reduced. Note that assigning neurons to nodes in such a manner here will not significantly change the number of spike events communicated between nodes – each population of the microcircuit forms connections with all others (with few exceptions; see Table 6.4).

The average number of spike events propagated between nodes in a simulation time step can be calculated from the population sizes N_X and the average firing rates $\bar{\nu}_X$ of the neurons in the populations by

$$\bar{\nu}_k = h \sum_X \bar{\nu}_X N_X \quad \forall \; X \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E,I}\}. \tag{6.8}$$

For the microcircuit model, this calculation yields $\bar{\nu}_k \approx 25$.

If workloads are balanced, i.e., if they do not vary much between nodes, then a node's excitatory and inhibitory workloads can be estimated from the population-specific workloads as

$$\bar{U}^{\mathrm{M}}_{\mathrm{exc}} = \frac{1}{M} \sum_Y \bar{U}^{\mathrm{POP}}_{\mathrm{exc},Y} \quad \forall \; Y \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E,I}\}, \tag{6.9}$$

$$\bar{U}^{\mathrm{M}}_{\mathrm{inh}} = \frac{1}{M} \sum_Y \bar{U}^{\mathrm{POP}}_{\mathrm{inh},Y} \quad \forall \; Y \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E,I}\}. \tag{6.10}$$

In Equations (6.9) and (6.10), M denotes the number of nodes given by $M = N/N^{\mathrm{M}}$, where N is the number of neurons simulated, and N^{M} specifies the number of neurons per node.

The total workload of a node is then given by

$$\bar{U}^{\mathrm{M}} = \bar{U}^{\mathrm{M}}_{\mathrm{exc}} + \bar{U}^{\mathrm{M}}_{\mathrm{inh}}. \tag{6.11}$$

Based on the population-specific workloads of the microcircuit (Table 6.5), the number of neurons in the network $N = \sum_X N_X = 77,169$, and the number of neurons per node $N^{\mathrm{M}} = 4096$ (defined as a node parameter; see Table 6.1), we derive the following workload estimates expressed in average number of synapse updates per simulation time step and node:

$$\bar{U}^{\mathrm{M}}_{\mathrm{exc}} = 2890, \quad \bar{U}^{\mathrm{M}}_{\mathrm{inh}} = 2242, \quad \bar{U}^{\mathrm{M}} = 5132.$$

The above calculation takes into account the specific connectivity between populations and the different average firing rates within populations. For $\bar{U}^{\mathrm{M}}$, the same value can be derived by assuming uniform connectivity across populations, which simplifies the calculation.

Workload estimation assuming uniform connectivity

We can express the assumption of uniform connectivity by an average mean out-degree calculated over all neurons in the network as

$$\bar{K}_{\text{out}} = \frac{K}{N}, \tag{6.12}$$

where K denotes the total synapses count in the network, and N is the number of neurons. It should be noted that $\bar{K}_{\text{out}}$ is also included in the Equations (6.5) and (6.6), but considered with its population-specific values as $\bar{\nu}_X K_{XY} = \bar{\nu}_X N_X (K_{XY}/N_X)$.

According to the workload definition, we can formulate the workload of a node for uniform connectivity as

$$\bar{U}^{\text{M}}_{\text{uni}} = \bar{C}^{\text{M}} \bar{\nu}_{\text{k}}, \tag{6.13}$$

where $\bar{C}^{\text{M}}$ denotes the average number of postsynaptic targets per presynaptic neuron and node, and $\bar{\nu}_{\text{k}}$ is the average number of spike events processed per simulation time step k.

By adapting Equation (5.1) from the workload model (see Section 5.2.1) using the relationship $\bar{K}_{\text{out}} = \epsilon N$, we can derive $\bar{C}^{\text{M}}$ by

$$\bar{C}^{\text{M}} = \bar{K}_{\text{out}} \frac{N^{\text{M}}}{N}, \tag{6.14}$$

where N^{M} specifies the number of neurons per node. Note that the workload model is applied here to network sizes of $N > 10^5$, for which a uniform connection probability ϵ across the network is now assumed.

From the Equations (6.12) to (6.14) and using Equation (5.2) we derive

$$\bar{U}^{\text{M}}_{\text{uni}} = h \bar{\nu} K \frac{N^{\text{M}}}{N}, \tag{6.15}$$

where h is the simulation resolution, and $\bar{\nu}$ is the average firing rate over all neurons in the network, which can be calculated from the population firing rates and population sizes by

$$\bar{\nu} = \frac{1}{N} \sum_X \bar{\nu}_X N_X \quad \forall\, X \in \{\text{L2/3, L4, L5, L6}\} \times \{\text{E,I}\}. \tag{6.16}$$

For the microcircuit model, this results in a calculated average firing rate of $\bar{\nu} = 3.24$ spks/s.

The microcircuit has a total synapse count of $K = 298,880,941$ and comprises $N = 77,196$ neurons. Using the Equation (6.15) and the defined set of node parameters (see Table 6.1) with $h = 0.1$ ms and $N^{\text{M}} = 4096$, the estimated workload of a compute node for an assumed uniform connectivity yields $\bar{U}^{\text{M}}_{\text{uni}} = 5137$. This is almost the same result as when population connectivity

is included, i.e., it is $\bar{U}^{\mathrm{M}} \approx \bar{U}^{\mathrm{M}}_{\mathrm{uni}}$.

This finding is unexpected and implies that for the microcircuit model the following relation holds

$$\bar{\nu}K \approx \sum_{Y}\sum_{X} \bar{\nu}_X K_{YX} \quad \forall\, X, Y \in \{\mathrm{L2/3, L4, L5, L6}\} \times \{\mathrm{E,I}\}. \tag{6.17}$$

I can only speculate that this property of the microcircuit model arises from specific connectivity determining the features of network activity, rather than being so dependent on synaptic weights.

This finding can be exploited to derive a workload estimate for the multi-area model.

Multi-area model

The multi-area model connects 32 microcircuits (256 populations), each representing one of 32 vision-related areas within one hemisphere of the macaque cortex. The model adapts the population sizes of the microcircuits for each area and the area's local connectivity, and adds a cortico-cortical connectivity, the connectivity between the areas. The multi-area model consists of 4.1 million neurons and 24 billion synapses.

Although the model adapts the microcircuits, there is no need for complex calculations to derive a reasonable workload estimate. We can use Equation (6.15) here. This is justified for the following reasons: for local connectivity, population pairs have the same relative in-degrees as in the microcircuit model; and the relative amount of local synapses is constant across areas (see Schmidt et al., 2018a). The model here *"preserves a defining characteristic of the local circuit"* (Schmidt et al., 2018b). We can therefore assume that the observed property expressed by Equation (6.17) is also preserved for the up-scaled microcircuits, and thus for the multi-area model. This also includes average firing rates; both models feature a biologically realistic network activity. Thus, we can extrapolate the workload that the multi-area model imposes on a compute node from the characteristics and calculated workloads of the microcircuit model using Equation (6.15).

For the multi-area model, with a network size of $N = 4.1 \cdot 10^6$ neurons, a total synapse count of $K = 24 \cdot 10^9$, an average firing rate of $\bar{\nu} = 3.24$ spks/s, and the defined set of node parameters (see Table 6.1), the estimated workload of a compute node results in

$$\bar{U}^{\mathrm{M}} = 7768.$$

This workload value falls within the range expected here. Although both models feature a natural dense connectivity, approximately 50% of the synaptic inputs of the microcircuit are external inputs, which do not account for workload. In case of the multi-area model, there are

	Microcircuit Model	Multi-Area Model
Excitatory, $\bar{U}^{\mathrm{M}}_{\mathrm{exc}}$ (65,3%)	2890	5073
Inhibitory, $\bar{U}^{\mathrm{M}}_{\mathrm{inh}}$ (43,7%)	2242	3395
Total, $\bar{U}^{\mathrm{M}}$	5132	7768

Table 6.6 | Estimated workloads. Average number of synapse updates that a compute node has to process per simulation time step when simulating the microcircuit model and the multi-area model, respectively. The values have been estimated for the set of node parameters listed in Table 6.1, i.e., a simulation resolution of $h = 0.1$, and a number of neurons per node of $N^{\mathrm{M}} = 4096$.

approximately 32% external inputs to each neuron in the network (see Schmidt et al., 2018a). The higher workload of the multi-area model is hence a consequence of the increased in-degrees. The magnitude of the increase in workload is therefore plausible, as it corresponds to the reduction in external inputs.

Unlike the microcircuit model, when simulating the multi-area model, the assignment of neurons to nodes affects the number of spikes communicated between nodes. An area only connects to a subset of areas. Therefore, the populations of an area should be kept close together, when assigning neurons to nodes. Optimizing spike communication between nodes is an aspect of network generation. However, it is not discussed further here, as it is not relevant to a node's workload – at least as long as workload is balanced among nodes.

Table 6.6 summarizes the estimated workloads for both models. For the excitatory and inhibitory workload components of the multi-area model, it was assumed that their ratio is also a property of the microcircuit model that is preserved. Their values were calculated accordingly based on the total workload.

These workload estimates are used in the next section to investigate the effectiveness of the proposed architecture enhancements.

6.3.3 Achievable Acceleration and Required Memory Bandwidth

The proposed enhancements and architecture alternatives differ in their effectiveness. This is analyzed in the following by examining the achievable acceleration factors with respect to workload and degree of processing parallelism. Specifically considered is the number of RBUs and whether excitatory and inhibitory spike events are processed simultaneously or in a serial fashion. Also determined is the required memory bandwidth for a given degree of parallelism and workload.

The following RBB architecture variants are considered:

- without memory partitioning (Figure 6.3A),
- with memory partitioning and interleaved RB memory access (Figure 6.3B), and
- with memory partitioning and parallel RB memory access (Figure 6.3C and D).

Not considered is the parallelization option in which connections with short transmission delays are stored on-chip (see Figure 6.1B). The impact on performance is expected to be small compared to parallelizing the processing using multiple RBUs and is also difficult to assess without precise knowledge of the memory technology used. Therefore, this option is not further investigated.

The average number of clock cycles that an RBU requires per simulation time step to process a given workload can be calculated by

$$L_{\mathrm{RBU}} = \mathrm{II}\frac{\bar{U}^{\mathrm{M}}_{\mathrm{eff}}}{n_{\mathrm{RBU}}} + L_{\mathrm{rw,seg}}, \quad \text{with} \tag{6.18}$$

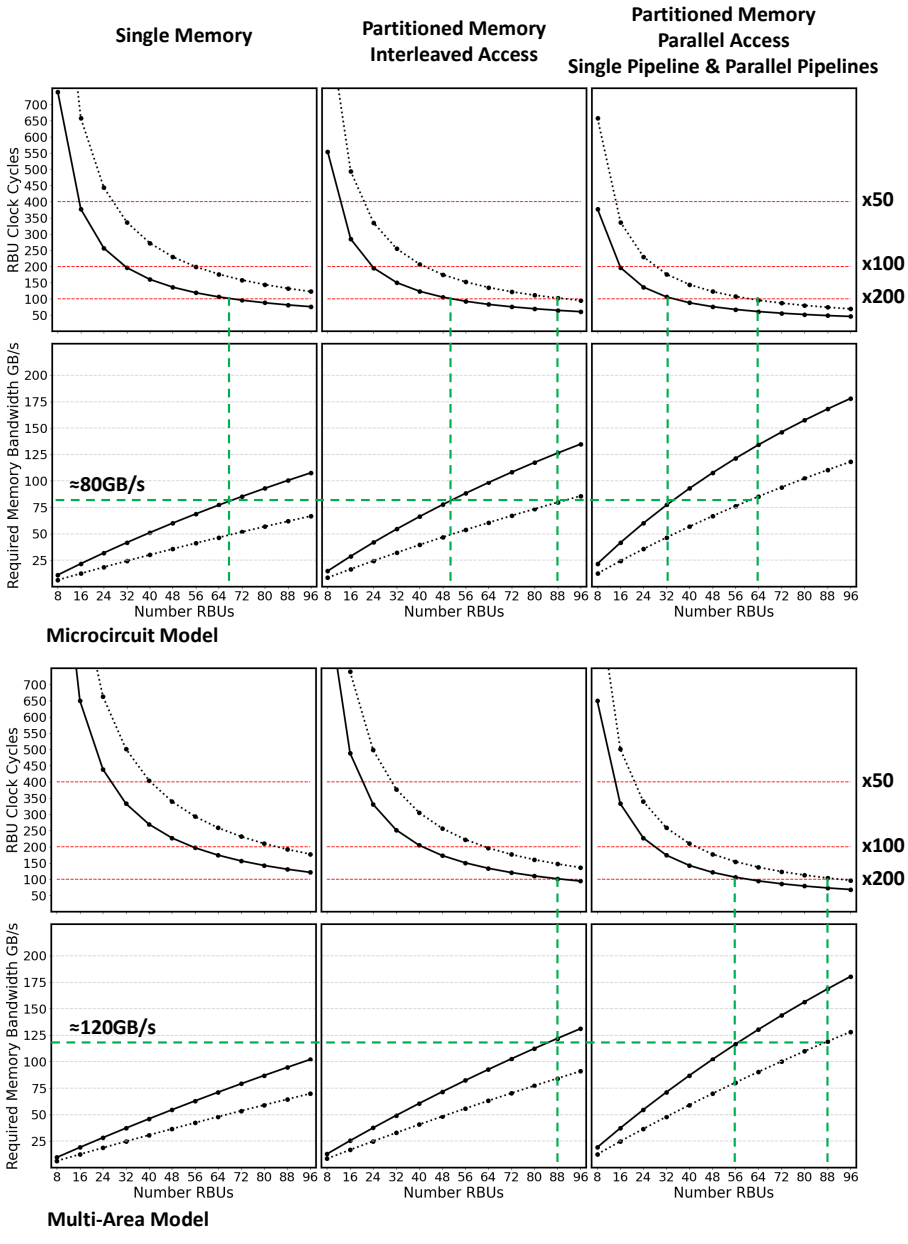
$$\bar{U}^{\mathrm{M}}_{\mathrm{eff}} = \begin{cases} \bar{U}^{\mathrm{M}}_{\mathrm{exc}} + \bar{U}^{\mathrm{M}}_{\mathrm{inh}} & \text{if spike processing is serialized} \\ \bar{U}^{\mathrm{M}}_{\mathrm{exc}} & \text{otherwise.} \end{cases} \tag{6.19}$$

In Equation (6.18), II denotes the initiation interval achieved by a specific RBB architecture variant, n_{RBU} represents the number of parallel RBUs, $L_{\mathrm{rw,seg}}$ corresponds to the RB segment read/write latency (see Table 6.2), and $\bar{U}^{\mathrm{M}}_{\mathrm{eff}}$ specifies a node's *effective workload*. It considers whether a specific architecture variant processes excitatory and inhibitory spike events serially or in parallel, as expressed by Equation (6.19). Architecture variants with simultaneous spike processing feature two RBBs per RBU (see Figure 6.1A), one RBB for each type of events. For these architectures, RBU latency is determined by the RBB that has to process the higher workload, which is the excitatory workload (see Table 6.6). In contrast, for the architecture variants with serialized spike processing, the workload is the sum of the excitatory and inhibitory workloads.

In order to achieve the best possible performance, the available memory bandwidth must be sufficiently large to serve the RBUs. The required memory bandwidth here depends on the RB pipeline throughput, the degree of parallelism, and the workload. It can be calculated as

$$B_{\mathrm{RBU,total}} = \frac{f_{\mathrm{clk}}}{L_{\mathrm{RBU}}}(\bar{U}^{\mathrm{M}}_{\mathrm{exc}} + \bar{U}^{\mathrm{M}}_{\mathrm{inh}})w_{\mathrm{len,LST}}, \tag{6.20}$$

where f_{clk} denotes the clock frequency at which the RBUs operate, and $w_{\mathrm{len,LST}}$ is the word length of an LST element.



Caption overleaf.

Figure 6.8 | Number of RBU clock cycles and required memory bandwidth for different architecture variants under large-scale workloads. Comparison of the architecture variants corresponding to Figure 6.3 with respect to achievable acceleration, required memory bandwidth, and degree of parallelism (number of RBUs). For different numbers of RBUs, the number of RBU clock cycles per simulation time step and the required memory bandwidth to process the workload generated by the microcircuit model (upper panels) and the multi-area model (lower panels) are shown. The black solid lines indicate architecture variants that simultaneously process excitatory and inhibitory spike events, while the black dashed lines represent those that do not. The red dashed lines mark the number of clock cycles corresponding to acceleration factors of 50, 100, and 200, relative to a clock frequency of 200 MHz. The required memory bandwidths are calculated assuming a 64-bit word length for an LST element. The green dashed lines represent the required memory bandwidth to achieve an RBU acceleration factor of 200 for both the microcircuit model (upper panels) and the multi-area model (lower panels), and indicate the number of RBUs that the architecture variants must implement to process the workload.

Figure 6.8 compares the different architecture variants based on the estimated workloads (see Table 6.6). Shown is the number of RBU clock cycles as a function of the number of RBUs, according to Equations (6.18) and (6.19), along with the corresponding required memory bandwidth (Equation (6.20)) for the workloads generated by the microcircuit model (upper panels) and the multi-area model (lower panels). The black solid lines represent architectures that simultaneously process excitatory and inhibitory spike events, while the black dashed lines represent those that do not. The red dashed lines mark the number of clock cycles that correspond to acceleration factors of 50, 100, and 200, relative to a clock frequency of 200 MHz. Note that acceleration here solely refers to RBU performance and does not account for latencies from neuron state propagation, spike communication, or inter-node synchronization. Assuming these latencies sum to 500 ns, the RBUs must be capable of delivering an acceleration factor of 200 to achieve a system-level acceleration of 100. All synaptic events must therefore be processed within 100 clock cycles.

The green dashed lines mark the required memory bandwidths necessary to achieve an RBU acceleration factor of 200 for the microcircuit model (upper panels) and the multi-area model (lower panels), and also indicate the number of RBUs that the architecture variants must implement to process the workload. The bandwidths are calculated for a clock frequency of $f_{\mathrm{clk}} = 200$ MHz and an LST element word length of $w_{\mathrm{len,LST}} = 64$ bit.

While all architecture variants can cope with the workload generated by the microcircuit and achieve high acceleration, for the workload of the multi-area model, only the variants that use memory partitioning are capable of achieving an RBU acceleration factor of 200. Nevertheless, a high degree of processing parallelism is required here. For example, 56 RBUs (112 RBBs) are required for the partitioned memory variant with parallel memory access and the simultaneous processing of excitatory and inhibitory spike events.

The memory bandwidth requirements are demanding. To achieve an overall acceleration factor

of 100, about 80 GB/s of memory bandwidth is required for the simulation of the microcircuit model. The multi-area model requires approximately 120 GB/s. However, recent FPGA-SoC technology is capable of providing even higher memory bandwidths. An example of such a device is given in the discussion in Section 6.5.

6.4 System Integration

The computational neuroscience community has developed a variety of software tools, toolchains, and workflows for neural network modeling, simulation, and analysis. To achieve high user acceptance and to make neuromorphic computing a useful tool for neuroscientists, an NC system must integrate well with this infrastructure. Moreover, to perform the compute-intensive simulation tasks, often high-performance computing (HPC) systems are used, where the trend goes toward heterogeneous systems and a *Modular Supercomputing Architecture* (MSA) – a concept developed at the Jülich Supercomputing Centre (Suarez et al., 2019, 2021). The idea of the MSA concept is to provide and integrate components tailored to the different computational behaviors of parts of a complex task or workflow to make computing more cost-effective. Rather than operating as a standalone system, it is therefore desirable for an NC system to also be integrated into this landscape.

In a large-scale simulation experiment, there is a number of tasks performed (e.g., pre- and post-processing steps) that can benefit from such heterogeneous architecture. Before a network can be simulated, it needs to be generated. For large networks, this is a compute-intensive process that creates a vast amount of connectivity data. For example, assuming a 64-bit value is used to represent a synaptic connection, the connectivity data of the multi-ara model comprising 24 billion synapses amounts to approximately 192 GB of data. Also computationally intensive is the analysis of simulation data. One example is the calculation of correlations between spike trains (see Section 2.3.2.2). These are just two exemplary tasks that can be conveniently performed on traditional compute clusters. They are less suitable – or even unsuitable – for execution on neuromorphic compute nodes, which are specifically designed to accelerate the simulation itself. Furthermore, a simulation must also be set up, which involves administrative tasks such as operation control and data management.

A possible concept for coupling an NC system with an HPC system, which separates these concerns, is shown in Figure 6.9. The HPC system here is intended to be used to perform simulation pre- and post-processing steps and to execute complex workflows. In order to be able to transfer the data to and from the NC system in an efficient way, the coupling of the two systems requires a high-bandwidth connection. A set of conventional compute nodes – here,

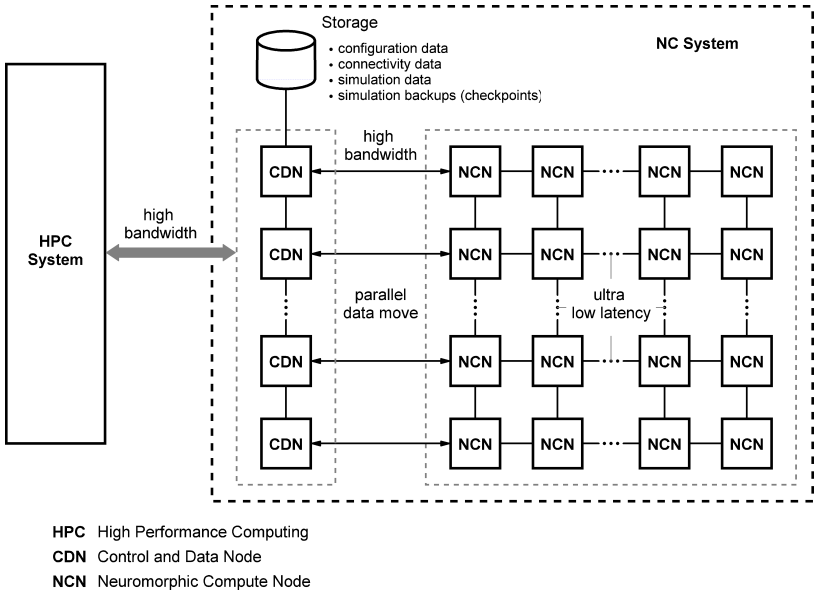


Figure 6.9 | High-level architecture concept of an HPC and NC system integration. A cluster of neuromorphic compute nodes (NCNs) is coupled to an HPC system through a number of control and data nodes (CDNs). This architecture concept considers the different concerns regarding the computational behavior and technical requirements of complex simulation tasks and workflows. Simulation pre- and post-processing can be performed conveniently on the HPC system. Data management and simulation control is delegated to the CDNs. The cluster of NCNs performs the simulation. See also the main text for description.

called control and data nodes (CDNs) – connect the HPC system to a cluster of neuromorphic compute nodes (NCNs). The CDNs are intended for local data management and for controlling the operation of the accelerator, for setting up the NCNs, as well as for monitoring simulations and collecting simulation data, such as spike recordings. Another use case of the CDNs can be *checkpointing* – that is, saving an interim state of a simulation in order to resume this simulation at this *checkpoint* at a later time. To enable a fast setup of the NCNs, the CDNs also allow to parallelize the configuration process by means of a *parallel data move* strategy. Network instantiation – the allocation of hardware resources on a node – is then also performed in a parallel fashion, node-local on the NCNs.

While high bandwidth is required to efficiently exchange large amounts of data with the HPC system, ultra-low latency interconnects within the cluster of NCNs are essential for simulation performance to achieve high acceleration. Here, dedicated communication networks tailored to

the respective task were proposed for the HNC node architecture, separating spike communication and inter-node synchronization (not shown in Figure 6.9; see also Section 3.3.4.8). In addition, to support forms of plasticity that are globally modulated, one could also imagine a separate communication network dedicated to plasticity.

Finally, it is worth noting that also the integration of the NC components into the HPC software infrastructure (e.g., the user and resources management of an HPC system) is crucial, in addition to the physical coupling. Providing a convenient, user-friendly system access fosters usability and user acceptance.

6.5 Discussion

Scaling up system size to enable the simulation of large-scale networks consisting of hundreds of thousands or even millions of neurons imposes additional design constraints. The higher workload placed on a compute node requires a higher degree of processing parallelism to keep latencies low and also entails a significant increase in the required memory bandwidth. Some of the decisions made for the HNC node architecture have therefore been reconsidered. A workload analysis based on two popular neuroscience models – the microcircuit model and the multi-area model – showed that in a realistic large-scale scenario, approximately 8000 synapse updates per simulation time step need to be performed by a single compute node capable of simulating 4096 neurons.

Based on the HNC node design presented in Chapter 3, proposals for additional parallelization options and alternative RB architecture designs utilizing partitioned memory were presented. A performance estimate showed that these architectural improvements are effective in providing significant acceleration, even under the high workloads expected for a large-scale system. However, the scale-up to the intended system size is a demanding technical challenge. Assuming a node specification according to the defined set of node parameters listed in Table 6.1, 1000 compute nodes are required to simulate the multi-area model. If we further assume that latencies for neuron state propagation, spike communication, and inter-node synchronization do not exceed 500 ns, a compute node must implement a minimum of about 56 RBUs to reach an acceleration factor of 100. The required memory bandwidth for this setup then results in approximately 120 GB/s. For comparison, the available memory bandwidth provided by the Zynq-7000 device used to implement the HNC node prototype was measured to be 1.862 GB/s (see Section 3.3.4.2).

In the pursuit of high simulation acceleration, memory technology, in terms of memory size, latency, and bandwidth, is a critical aspect of the design. Recent advances in FPGA-SoC technology hold promise here. For example, the AMD Xilinx Versal HBM adaptive SoC series

(AMD Xilinx, 2023b) is specifically designed for memory-bound, compute-intensive applications and integrates high-bandwidth memory (HBM) technology. This technology enables up to 819 GB/s memory bandwidth for these devices, with memory sizes of up to 32 GiB. Such performance is achieved through an integration of the DRAM memory dies with the SoC device die using a silicon interposer (AMD Xilinx, 2023a). The devices also provide a large amount of programmable logic resources, on-chip memory, and a powerful ARM-based processing system (AMD Xilinx, 2024).

Scaling the HNC node design to the size discussed here – i.e., a degree of processing parallelism that enables a speed-up factor on the order of 100 in a large-scale simulation – would require 2576 DSP blocks to implement the ODE solver pipelines of the 56 RBUs (112 RBBs) needed here. The Versal HBM adaptive SoC series devices provide up to 10,848 DSPs (AMD Xilinx, 2024), which is more than four times the number of resources required.

With recent commercially available FPGA-SoC and HBM technology, a large-scale neuromorphic computing system that can deliver significant acceleration while meeting the demanding requirements of neuroscience simulation becomes within reach; although building such a system remains a demanding technical challenge. The integration of neuromorphic computing into the HPC landscape – linking hard- and software components at the system level – can here enable a much broader spectrum of use cases than a standalone NC system. It will allow neuromorphic computing to be included in complex workflows, for example, those performed in multiscale co-simulations (Kusch et al., 2022). Furthermore, the FPGA-SoC technology is highly adaptable. Although the proposed concept aims at a large-scale NC system and research facility intended as a tool for neuroscientists, such a system can also provide a flexible hardware platform to accelerate applications from other domains with similar requirements.

Chapter 7

Discussion and Outlook

7.1 Conclusions

Progress in neuroscience research depends to a large extent on the ability to study large neural networks and perform complex simulations. Performing simulations in hyper-real time is of great interest here, as it would allow comprehensive parameter scans and the study of slow processes such as learning and long-term memory. Even the fastest supercomputers available today cannot meet the challenge of significantly accelerating the simulation of a large-scale network. Neuromorphic computing, leveraging novel technologies and application-specific hardware architectures, is therefore an attractive option that promises to provide the necessary tools for this task.

Despite all the technological innovations, making neuromorphic computing a useful tool for neuroscientists is a demanding technical challenge. Neuroscience research employs mathematical models to gain understanding of the complex dynamics of neural networks. Their simulation requires numerical accuracy, and simulation outcomes must be deterministic and reproducible. The plethora of models demands a high degree of flexibility for their algorithmic implementation. In addition, to bring neuromorphic computing to users, the existing infrastructure and complex landscape of tools for modeling and simulation must also be taken into account.

On a technical level, flexibility and efficiency are conflicting goals, making it difficult to satisfy all requirements equally. Traditional general-purpose processors offer a high degree of flexibility, but their efficiency is low. Physically optimized ASICs achieve high efficiency, but are inflexible and chip development is expensive. Commercial off-the-shelf FPGA-SoC technology offers a good compromise here. The tight coupling of general-purpose processors with a programmable logic device in a single chip allows an application to combine the flexibility of a software-based solution with the efficiency of application-specific hardware. In this regard, the HNC node's hybrid hardware and software mixed architecture can provide a level of flexibility that allows the system to adapt to changing requirements and keep pace with the rapid developments in neuroscience, while delivering significant acceleration.

The design space exploration, workload analysis, and performance evaluations that accompanied the development of the HNC node identified memory technology as a critical factor, imposing design constraints that largely determine system performance and size. Recent advances in FPGA-SoC technology, particularly the integration of HBM technology, are promising in this regard, as they address compute-intensive, memory-bound applications and enable high acceleration even for heavy workloads.

Commercial off-the-shelf FPGA-SoC technology has the potential to provide the substrate to take neuromorphic computing as a tool for neuroscience to the next level, without the need for

costly chip development. This potential arises from the convergence of computation, memory, and connectivity that the technology provides. Acceleration factors on the order of 100 are within reach, even for the simulation of large-scale spiking neural networks. However, building such a system, bringing it to the desired scale, and making it available to users remains a major technical and economic challenge.

7.2 Summary and Discussion

Neuroscience requirement-driven design

The design of the HNC node has been strictly driven by the requirements of computational neuroscience modeling and simulation. This distinguishes this development from other efforts in the field as a complementary yet distinct approach to the neuromorphic developments that aim at brain-inspired novel computing architectures for solving real-world tasks. The requirements are demanding. Numerical accuracy, flexibility in model implementation, and the ability to deliver significant acceleration are the key requirements that define the boundary conditions of the design.

Numerical accuracy and implementation correctness

Even for domain experts, it can be difficult to judge the correctness of a simulation outcome. Credibility can be built by formalizing processes. This applies to the modeling, implementation, and simulation tasks performed in an experiment. Although appropriate methods, such as verification and validation methodologies, exist, they are not yet well established in the field of neural network modeling and simulation. Here, a clear terminology is essential for effective communication. It avoids ambiguity and is a factor of quality.

A reasonable adaptation of the existing terminology for model verification and validation was proposed that is more explicit and better expresses the underlying intent in the field of computational neuroscience. The concept of *model verification and substantiation* was introduced as a methodology that allows for the accumulation of evidence of a model's plausibility and correctness, even in the absence of experimental validation data. Applying this method and following a systematic approach has proven to be of great value. It not only created a high level of credibility for correctness, but also guided design decisions. For example, the choices made with regard to numerical precision, data types, and algorithms are based on the results of a series of calculation verification tasks that have been conducted.

The selection of the test case model – the minimal two populations Izhikevich network model – can certainly be questioned here, as the model is less relevant to the field. I have argued that the model contains a number of non-standard features in its conceptual and implementational choices

that make it a particularly illustrative example for the demonstration of a rigorous verification and validation process. However, more importantly, this model places higher demands on a hardware implementation than, for example, a leaky integrate-and-fire (LIF) neuron model. In this respect, the requirements are defined more stringent than would typically be needed, which provides an additional level of confidence. In this context, the co-verification process that accompanied the hardware-software co-development of the HNC node must also be seen as a process designed to bring rigor to verification with the goal of building quality into implementations.

While *verification* ensures appropriate implementation, *validation* evaluates the consistency of the simulation outcome with the *system of interest*. By carrying out a substantiation assessment that compared the HNC node's simulation outcome with reliable references – testing the equivalence of statistical features – a high degree of credibility in correctness has been achieved.

Flexibility

The need for flexibility arises mainly from modeling. A plethora of neuron and synapse models exist and new models are being formulated. The hybrid hardware and software mixed architecture of the HNC node, its modularity, and the use of generic data types address this requirement and provide the technical prerequisites to make the system amenable to existing code generation techniques.

The HNC node architecture is open to extensions, such as the integration of plasticity algorithms. Plasticity rules and algorithms is a rapidly evolving area of research that requires a high degree of flexibility in algorithmic implementation. The HNC node stores synaptic weights as well as spike events in external memory accessible by both the application processing unit and the programmable logic part. This enables spike-based plasticity rules to be implemented in software and then run on a dedicated *plasticity processor*, which can be supported by hardware accelerators.

The FPGA-SoC technology here has an advantage over a pure FPGA solution because it integrates general-purpose processors in a very powerful setup, typically multi-cores combined with vectorization units and real-time processors. The ability to implement tasks in software also facilitates system integration. The HNC node takes advantage of this and provides a basic C-API.

Performance

The performance of tools and hardware employed for simulations is typically evaluated using dedicated benchmark models that serve as a reference and make results comparable. Deriving general statements about the behavior of a system from these benchmarks is difficult. A benchmarking approach that allows workloads, and thus compute costs, to be varied independent of network

size is of great value and can provide a more comprehensive view of a system's performance characteristics. Therefore, for the purpose of a systematic performance assessment of the HNC node, a workload model was developed that defines a node's *workload* as the average number of spike events processed per simulation time step. For network sizes up to 10^5 neurons, the model introduces a metric that is independent of the number of neurons simulated. In addition, an accurate performance model has been formulated for the HNC node that, although specific to the HNC node architecture, captures in a general way the performance-determining aspects of the hybrid time- and event-driven scheme used in digital simulations of spiking neural networks.

Accompanied by performance measurements, the performance model allowed a verification of the implementation in terms of expected performance, as well as it quantifies the performance-determining processes. For the single node prototype, this approach also allowed to simulate workloads that are to be expected for cluster operation. The performance characteristics of the single HNC node, as well as when operated in a cluster, could thus be predicted for varying assumptions regarding workload and hardware design choices. It could be shown that depending on the workload situation and the design goal, different hardware setups may be appropriate to achieve best performance.

Scalability

Scaling up the system size to enable the simulation of large-scale networks imposes additional design constraints. The higher workload requires a higher degree of processing parallelism to retain system performance, and also entails a significant increase in the required memory bandwidth. This has led to the introduction of additional parallelization options and the reconsideration of some design decisions regarding the architecture of the ring buffer memories. These architectural enhancements have been evaluated for their ability to deliver significant acceleration at large-scale workloads. The definition of these workloads was derived from an analysis of the connectivity and firing statistics of two biologically realistic neural network models with natural dense connectivity.

The proposed enhancements showed to be effective in achieving an acceleration factor on the order of 100. Once again, memory technology plays a critical role here. The memory bandwidth requirements are demanding and have been estimated at approximately 120 GB/s. However, recent FPGA-SoC technology is capable of providing this bandwidth.

System integration

To make neuromorphic computing a useful tool for computational neuroscience, a neuromorphic computing system must integrate well with the existing landscape of tools and workflows. This

includes the HPC landscape used by computational neuroscientists to perform the compute-intensive simulations. Therefore, this thesis also briefly discussed aspects of system integration and presented a basic high-level architecture concept for an HPC integration.

7.3 Outlook

The landscape of neuromorphic computing is broad, and even subareas such as the field of computational neuroscience modeling and simulation, which is the subject of this thesis, already span a wide range of topics and aspects. Therefore, not all ideas and concepts presented in this work are fully developed. Some questions remain unanswered, require further investigation, and may even change in the rapidly evolving field.

I deliberately left out the complex topic of plasticity rules and algorithms, although their requirements were taken into account in the design of the HNC node. The computational cost of these algorithms and their exact impact on performance is unclear, but is expected to be significant. Synaptic plasticity changes synaptic weights, structural plasticity creates and deletes synaptic connections. Both change the connectivity data of a network and cause additional memory read and write operations. This makes it all the more important that a neuromorphic architecture can deliver a high base acceleration and being prepared for this shift in computational cost. Here, the HNC node can serve as a practical and convenient platform for hardware prototyping to explore plasticity algorithms and to find efficient implementations.

The ability of the proposed architecture to scale to larger system sizes was evaluated based on a performance model, which was derived from the microarchitecture of the prototypical implementation. Although this model has been shown to accurately describe the performance characteristics of a single HNC node, this approach leaves an element of uncertainty because it makes simplifying assumptions about the inter-node communication in a cluster. Ultra-low latency solutions for efficient spike packet communication exist (see the introduction Section 1.3.3 and the discussion in Section 3.4), but need to be proven practical and effective for the HNC node architecture.

Building a large-scale neuromorphic system – even when based on commercial off-the-shelf technology – requires sustantial technical and economic effort. However, the vast majority of current spiking neural network modeling studies use models at the scale of up to a few tens of thousands of neurons. For these applications, a smaller neuromorphic system comprising a few tens of nodes and capable of delivering significant acceleration would also be attractive. Such a system could also serve as an ideal platform and practical tool for exploring algorithms and novel

architectures, and would be an intermediate step toward a next-generation neuromorphic system and neuroscience simulation platform.

Appendix A

Two-Population Izhikevich Network Model

Tables A1 and A2 provide a comprehensive description of the minimal two-population Izhikevich neural network model. The model is used in Chapter 2 to demonstrate a rigorous model verification and substantiation workflow, in Chapter 4 to verify and validate the correctness of the HNC node software and hardware implementation, and in Chapter 5 to examine the HNC node performance characteristics.

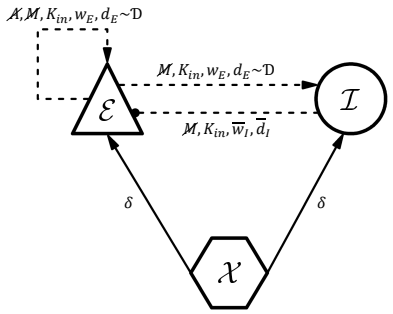
The neuron model by Izhikevich was originally published in Izhikevich (2003), the two-population network model in Izhikevich (2006).

The format of the tabular and graphical representation of network connectivity, network parameters, and simulation setup follows the proposed methods described in Nordlie et al. (2009) and Senk et al. (2022).

A.1 Network Description

Summary		
Populations	excitatory population, inhibitory population	
Connectivity	random, independent with fixed in-degrees, respecting Dale's principle	
Neuron model	Izhikevich neuron model, regular-spiking and fast-spiking neuron model type	
Synapse model	plastic synaptic weights (static for reproduction of network state), fixed delays	
Input	random input	
Populations		
Name	**Elements**	**Size**
$\mathcal{E}$	Izhikevich, regular-spiking	$N_{\mathrm{E}} = \beta N$
$\mathcal{I}$	Izhikevich, fast-spiking	$N_{\mathrm{I}} = N - N_{\mathrm{E}} = N - \beta N$

Connectivity		
Source	**Target**	**Pattern**
$\mathcal{E} \cup \mathcal{I}$	$\mathcal{E}$	random, independent, fixed in-degree $K_{\text{in}} = \epsilon N$, autapses and multapses prohibited ($\not{A}$, $\not{M}$), excitatory connection weight w_{E}, distributed excitatory connection delay $d_{\text{E}} \sim \mathcal{D}$, constant inhibitory connection weight $\bar{w}_{\text{I}}$, constant inhibitory connection delay $\bar{d}_{\text{I}}$
$\mathcal{E}$	$\mathcal{I}$	random, independent, fixed in-degree $K_{\text{in}} = \epsilon N$, multapses prohibited ($\not{M}$), constant inhibitory connection weight $\bar{w}_{\text{I}}$, constant inhibitory connection delay $\bar{d}_{\text{I}}$
$\mathcal{X}$	$\mathcal{E} \cup I$	one-to-one δ, external input i_{ext}



Neuron	
Type	Izhikevich model
Description	• dynamics of membrane potential $v_i(t)$ ($i \in \{1, \ldots, N\}$) $\frac{dv_i}{dt} = 0.04v_i^2 + 5v_i + 140 - u_i + I_i(t)$ with $I_i(t) = i_{\text{exc}}(t) + i_{\text{inh}}(t) + i_{\text{ext}}(t)$ $\frac{du_i}{dt} = a(bv_i - u_i)$ if $v_i \geq \theta$, then $\begin{cases} v_i \leftarrow c \\ u_i \leftarrow u_i + d \end{cases}$ • spike emission at t_k^i if $v_i(t_k^i) \geq \theta$ • initial values: $v_i(t = 0) = V_0$, $u_i(t = 0) = U_0$

Plasticity	
Type	additive spike-timing-dependent plasticity (STDP) rule
Description	excitatory connections are plastic according to the STDP rule: $w_{\mathrm{E}} \leftarrow \begin{cases} w_{\mathrm{E}} + A_+ \cdot \exp(-\Delta t/\tau_+) & : \Delta t \geq 0 \\ w_{\mathrm{E}} - A_- \cdot \exp(\Delta t/\tau_-) & : \Delta t < 0 \end{cases}$ • update rule: synaptic weight changes are buffered for one biological second and then the weight matrix is updated for all plastic synapses simultaneously
Input	
Type	input of a constant current into a single neuron
Description	input to the network is a constant current i_{ext} into a single neuron $n \in (E \cup I)$ randomly selected in the interval Δt_{ext}

Table A1 | Two-population Izhikevich network model description.

A.2 Network Parameters

Connectivity		
Name	**Value**	**Description**
N	1000	total number of neurons ($N_{\mathrm{E}} + N_{\mathrm{I}}$)
β	0.8	relative size of excitatory population
ϵ	0.1	connection probability
N_{E}	$\beta N = 800$	number of excitatory neurons
N_{I}	$N - N_{\mathrm{E}} = 200$	number of inhibitory neurons
K_{in}	$\epsilon N = 100$	number of synapses per neuron
Neuron		
Name	**Value**	**Description**
(a, b, c, d)	$(0.02, 0.2, -65.0, 8.0)$	model parameters: regular-spiking neuron type
(a, b, c, d)	$(0.1, 0.2, -65.0, 2.0)$	model parameters: fast-spiking neuron type
V_0	-65.0 mV	initial membrane potential $v_i(t = 0)$
U_0	$0.2V_0 = -13.0$ mV	initial value of recovery variable $u_i(t = 0)$
θ	30 mV	spike threshold
Synapse		
Name	**Value**	**Description**
w_{E}	6.0	initial excitatory synaptic weight (plastic)
w_{I}	-5.0	inhibitory synaptic weight
d_{E}	$[1, 2, ..., 20]$ ms	excitatory synaptic transmission delay, drawn from a uniform integer distribution
d_{I}	1 ms	inhibitory synaptic transmission delay

Plasticity		
Name	**Value**	**Description**
τ_+	20 ms	time constant, potentiation
τ_-	20 ms	time constant, depression
A_+	0.1 mV	amplitude, potentiation
A_-	0.12 mV	amplitude, depression
Input		
Name	**Value**	**Description**
i_{ext}	20 pA	external input current
Δt_{ext}	1 ms	external input current interval
Simulation		
Name	**Value**	**Description**
Δt	0.1 ms	time resolution

Table A2 | Two-population Izhikevich network model parameters.

Appendix B

Model Substantiation Assessment

To demonstrate the stability of the effect size measure in the data, the computed measures and effect sizes were collected from the network states after 1, 2, 3, 4, and 5 hours of simulation for visual inspection and comparison, corresponding to Figures B1-B5. The figures show the histograms (70 bins each) of the three characteristic measures computed from 60 seconds of network activity: left, firing rates (FR); middle, local coefficients of variation (LV); right, pairwise correlation coefficients (CC). For FR and LV, each neuron enters the histogram, for CC each neuron pair. Results are shown for three iterations (rows) of the substantiation process of the C model (dark colors) and SpiNNaker model (light colors). On the far right, the difference between the respective distributions is quantified by the effect size.

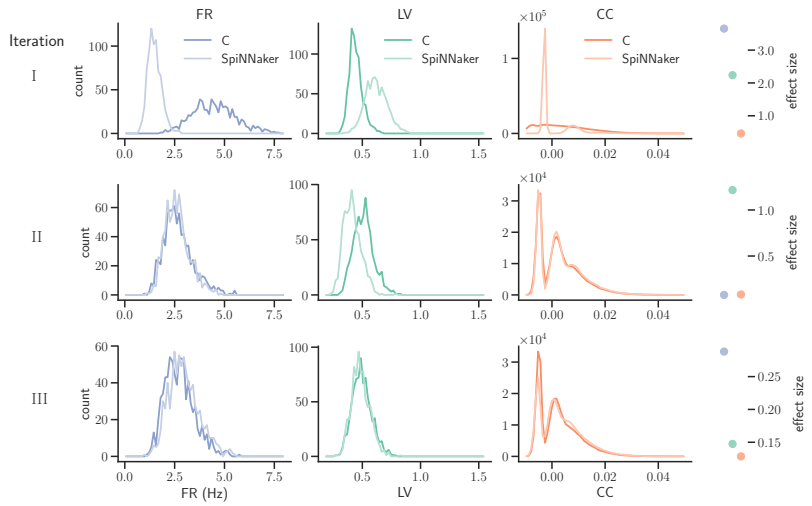


Figure B1 | Characteristic measures computed from 60 seconds of network activity after 1 hour of simulation.

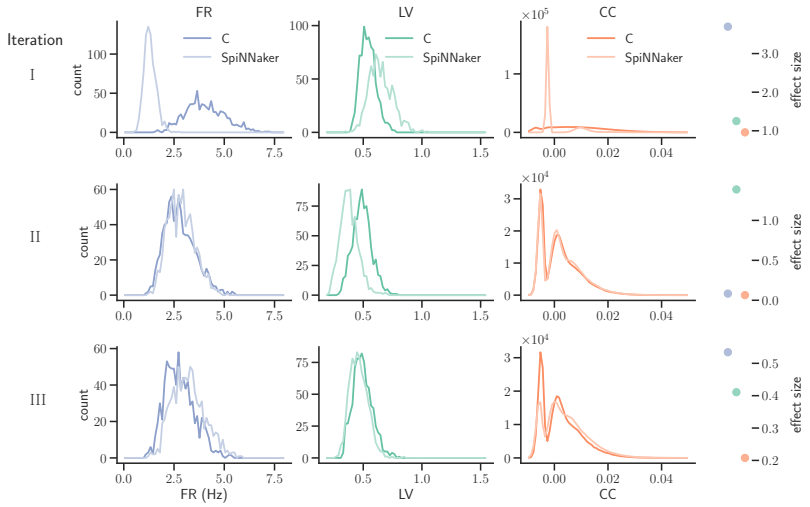


Figure B2 | Characteristic measures computed from 60 seconds of network activity after 2 hours of simulation.

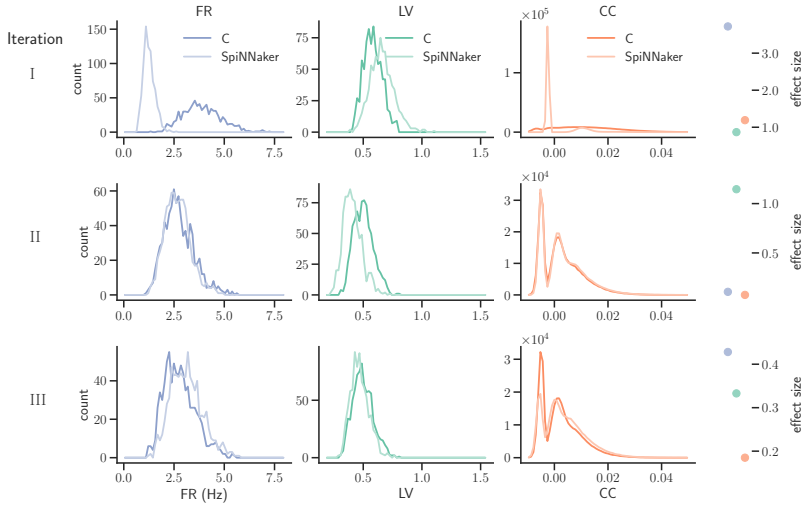


Figure B3 | Characteristic measures computed from 60 seconds of network activity after 3 hours of simulation.

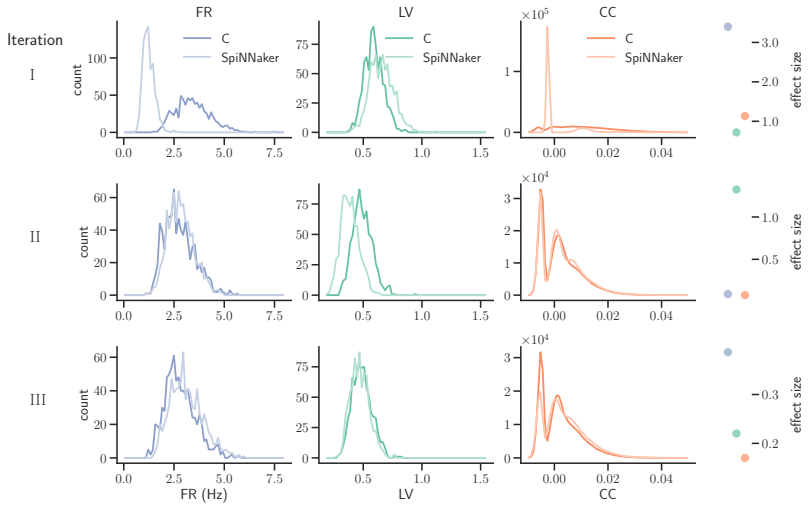


Figure B4 | Characteristic measures computed from 60 seconds of network activity after 4 hours of simulation.

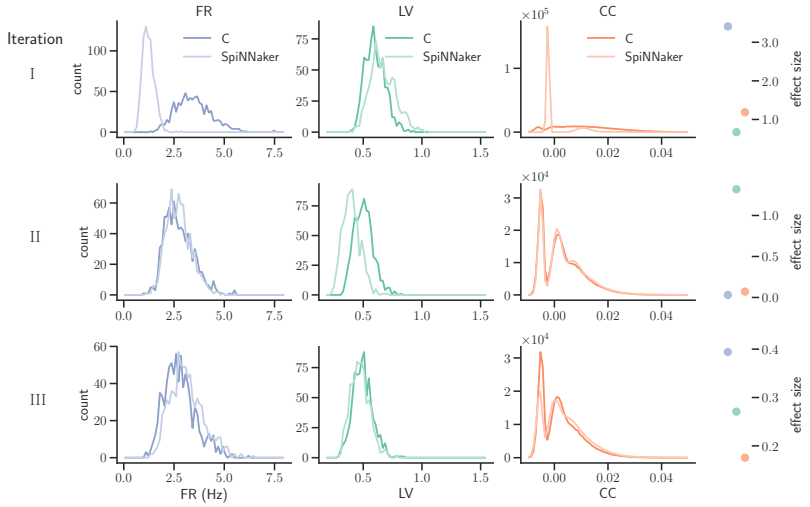


Figure B5 | Characteristic measures computed from 60 seconds of network activity after 5 hours of simulation.

Appendix C

Control Registers

Configuration and status information of the HNC node hardware blocks are stored in a number

Component	Register	Description	Access Type
Core Module FSM PRNG	SC_0	HNC node configuration, commands and status	read/write
	SC_1	Steps to simulate, k	write
	SC_2	Steps simulated	read
	SC_3	RB segment address (debug)	write
	SC_4	PRNG configuration	write
	SC_5	System information, P,N^{P}	read
	SC_6	Spike count	read
	SC_7	unused	read/write
PS/PL Data Transfer Module	TM_0	Configuration	write
	TM_1	HP1 read base address (S1)	write
	TM_2	HP1 write base address (read out of FIFOs for debug)	write
	TM_3	HP3 read base address (S2)	write
	TM_4	HP3 write base address (spike recording)	write
	TM_5	HPx read/write beats and bursts configuration	write
	TM_6	unused	read/write
	TM_7	FIFO select (debug)	write
Serializer Module	SM_0	Configuration	write
	SM_1	APU spike injection	write
	SM_2	unused	read/write
	SM_3	unused	read/write

Table C1 | HNC node configuration and status registers.

Appendix D

Connection Data Structure

The HNC node stores a network's connectivity by assigning each neuron in the network a list of the synaptic targets (LST) that the neuron has on the node. Each element in these lists encodes the synaptic weight, delay, and node-local id of the postsynaptic neuron for a single connection, using a 64-bit data structure. The data format is detailed in Table D1. The node-local id consists of a 6-bit value indexing the neuron's position in the pipeline and a 4-bit hardware control value s_i^{Sx} that determines the data path to the ring buffer (RB) and processing unit associated with the postsynaptic neuron. Setting $s_i^{\mathrm{Sx}} = b'1111'$ broadcasts an LST element to all RBs, enabling

Bits	**Data Type**	**Description**			
39:00	40-bit signed fixed-point s16.23	Synaptic weight, w_{ij}			
48:40	up to 9-bit unsigned integer	Synaptic delay, d_{ij}, in steps of $0.1\,ms$			
54:49	6-bit unsigned integer	Target neuron index within a processing unit, $\mathrm{n} = \{0, 1, .., N^{\mathrm{P}} - 1\}$, $N^{\mathrm{P}}_{\max} = 64$			
59:55	boolean	Control flags			
		Bit	*Description*		
		56	Reset RB segment		
		57	Set if $d_{ij} = D_{\min}$ (restart ODE solver pipeline flag)		
63:60	4-bit binary	Encodes the ring buffer FIFO index from which DMUX SEL is derived, s_i^{S1}, s_i^{S2}			
		Value	*Selected FIFOs*	*Processing Units*	
				Assigned to HP1 (S1)	*Assigned to HP3 (S2)*
		b'0000'	none	none	none
		b'1111'	F1, F2, F3, F4, F5, F6, F7, F8	P1, P3, P5, P7, P9, P11, P13, P15	P2, P4, P6, P8, P10 , P12, P14, P16
		b'0001'	F1	P1	P2
		b'0010'	F2	P3	P4
		...			
		b'1000'	F8	P15	P16

Table D1 | Synaptic target list item data structure.

Appendix E

Monte Carlo Simulations

To determine the throughput of the RBB architectures that use memory partitioning, as well as the FIFO buffer sizes required by the different architecture variants, models of their architectures were implemented in software in the C++ language. By means of Monte Carlo simulations, the achievable initiation intervals (see Section 6.2.2), the average number of RBU clock cycles, as well as the maximum utilization of the FIFO buffers were determined for different workloads and numbers of RBUs. The simulations were performed each with a simulation count of $1,000,000$. The tables below show the results of the Monte Carlo experiments for the architecture variants that serially process excitatory and inhibitory spike events.

E.1 Architecture Alternative: *Partitioned Memory Interleaved Access*

Workload, W	5000				10000			
Number of RBUs, n_{RBU}	16	32	64	128	16	32	64	128
Average RBU clock cycles, L_{RBU}	467	233	116	57	936	467	233	116
FIFO max utilization	126	66	36	19	241	125	67	36

Table E1 | Monte Carlo simulation results. RBB architecture: *Partitioned Memory Interleaved Access*. The architecture variant is shown in Figure 6.3B.

E.2 Architecture Alternative: *Partitioned Memory Parallel Access Parallel Pipelines*

Workload, W	5000				10000			
Number of RBUs, n_{RBU}	16	32	64	128	16	32	64	128
Average RBU clock cycles, L_{RBU}	324	164	83	42	643	324	164	83
FIFO max utilization	44	30	22	16	69	45	33	20

Table E2 | Monte Carlo simulation results. RBB architecture: *Partitioned Memory Parallel Access Parallel Pipelines* The architecture variant is shown in Figure 6.3C.

E.3 Architecture Alternative: *Partitioned Memory Parallel Access Single Pipeline*

Workload, W	5000				10000			
Number of RBUs, n_{RBU}	16	32	64	128	16	32	64	128
Average RBU clock cycles, L_{RBU}	333	171	88	46	655	333	171	88
FIFO 1 max utilization	84	60	42	29	130	85	59	40
FIFO 2 max utilization	91	62	44	29	125	82	61	46

Table E3 | Monte Carlo simulation results. RBB architecture: *Partitioned Memory Parallel Access Single Pipeline* The architecture variant is shown in Figure 6.3D.

Nomenclature

Acronyms

ACM	Association for Computer Machinery
APU	Application Processing Unit
ASIC	Application-Specific Integrated Circuit
ASIP	Application-Specific Instruction set Processor
AXI	Advanced eXtensible Interface
BRAM	Block RAM
CLB	Configurable Logic Block
COBA	COnductance-BAsed
CUBA	CUrrent-BAsed
DDR	Double Data Rate
DFE	DataFlow Engine
DMA	Direct Memory Access
DSL	Domain-Specific Language
DSP	Digital Signal Processor
DTM	Data Transfer Module
DUT	Device Under Test
EDA	Electronic Design Automation
eFPGA	embedded FPGA
EPSP	Excitatory PostSynaptic Potential
FPGA	Field Programmable Gate Array
FSM	Finite State Machine

GPP	General-Purpose Processor
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HBM	High-Bandwidth Memory
HLS	High-Level Synthesis
HNC node	Hybrid Neuromorphic Compute node
HP	High-Performance Port
HPC	High-Performance Computing
IPSP	Inhibitory PostSynaptic Potential
LFSR	Linear Feedback Shift Register
LIF	Leaky Integrate and Fire
LST	List of Synaptic Targets
LUT	Look-Up Table
MSA	Modular Supercomputing Architecture
NC	Neuromorphic Computing
NEST	NEural Simulation Tool
ODE	Ordinary Differential Equation
PL	Programmable Logic
PRNG	Pseudo-Random Number Generator
PS	Processing System
PSC	PostSynaptic Current
PSP	PostSynaptic Potential
RAM	Random Access Memory
RB	Ring Buffer
RBB	Ring Buffer Block
RBU	Ring Buffer Unit
RTF	Real-Time Factor

RTL	Register-Transfer Level
SCS	Society for Computer Simulation
SNN	Spiking Neural Network
SoC	System-on-Chip
SRAM	Static RAM
STDP	Spike-Timing-Dependent Plasticity
SVB	State Variables Buffer
UART	Universal Asynchronous Receiver Transmitter
UDP	User Datagram Protocol
VHDL	Very high speed integrated circuit Hardware Description Language

Symbols

$\bar{\nu}_k$	Average number of spike events per simulation time step
$\bar{\nu}$	Average number of spike events per second
$\bar{C}^{\mathrm{M}}$	Average number of postsynaptic targets per presynaptic neuron and node
$\bar{U}^{\mathrm{M}}_{\mathrm{eff}}$	Effective workload of a node
$\bar{U}^{\mathrm{M}}_{\mathrm{exc}}$	Excitatory workload of a node
$\bar{U}^{\mathrm{M}}_{\mathrm{inh}}$	Inhibitory workload of a node
$\bar{U}^{\mathrm{M}}$	Workload of a node: the average number of synapse updates per simulation time step a node has to perform
CC	Pairwise Pearson's correlation coefficient
CV	Coefficient of variation
FR	Firing rate
II	Initiation interval
LV	Local coefficient of variation
B	Bandwidth
$C^{\mathrm{M}}_{\mathrm{max}}$	Maximum number of postsynaptic targets per presynaptic neuron and node
C	Connection probability

D or d	Synaptic delay
f_{clk}	Clock frequency
F	Acceleration factor
G or g	Weighted synaptic input
h	Simulation resolution
J or w	Synaptic weight
K_{RB}	Number of ring buffer segments
K	Synapse count
k	Simulation time step
L	Latency
M	Number of nodes
N^{P}	Number of neurons per processing unit
N^{RB}	Number of neurons per ring buffer
n^{sp}	Number of spikes of a neuron
N	Number of neurons
n	Neuron number
P	Number of processing units or performance loss
S	Memory size
w_{len}	Word length

Units of Measurement

A	Ampere, measure of electric current
GB	Gigabyte, 1GB = 10^9 Bytes
GiB	Gibibyte, 1GiB = 2^{30} Bytes
Hz	Hertz, measure of frequency, expressed in s^{-1}
MB	Megabyte, 1MB = 10^6 Bytes
MB/s	Megabytes per second

MiB	Mebibyte, 1MiB = 2^{20} Bytes
MiB/s	Mebibytes per second
MOPS	Million operations per second
MT/s	Megatransfers per second

List of Figures

List of Tables

Bibliography

Abbott, L. F. and Regehr, W. G. (2004). Synaptic computation. *Nature* 431, 796–803. doi: 10.1038/nature03010

Akar, N. A., Cumming, B., Karakasis, V., Küsters, A., Klijn, W., Peyser, A., et al. (2019). Arbor — A Morphologically-Detailed Neural Network Simulation Library for Contemporary High-Performance Computing Architectures. In *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)* (Pavia, Italy: IEEE), 274–282. doi:10.1109/EMPDP.2019.8671560

Alfke, P. (1996). Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators (AMD Xilinx Application Note: XAPP 052) Available online at: www.xilinx.com

Altimus, C. M., Marlin, B. J., Charalambakis, N. E., Colón-Rodríguez, A., Glover, E. J., Izbicki, P., et al. (2020). The Next 50 Years of Neuroscience. *The Journal of Neuroscience* 40, 101–106. doi:10.1523/JNEUROSCI.0744-19.2019

AMD Xilinx (2018). Zynq-7000 SoC Data Sheet: Overview (DS190). Available online at: www.xilinx.com

AMD Xilinx (2019a). AXI DMA v7.1 LogiCORE IP Product Guide (PG021). Available online at: www.xilinx.com

AMD Xilinx (2019b). Embedded System Tools Reference Manual v2019.2 (UG1043). Available online at: www.xilinx.com

AMD Xilinx (2019c). Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator v2019.1 (UG994). Available online at: www.xilinx.com

AMD Xilinx (2019d). Vivado Design Suite User Guide High-Level Synthesis v2019.1 (UG902). Available online at: www.xilinx.com

AMD Xilinx (2019e). Vivado Design Suite User Guide v2019.1 (UG893). Available online at: www.xilinx.com

AMD Xilinx (2021a). UltraScale Architecture Memory Resources: User Guide (UG537) Available online at: www.xilinx.com

AMD Xilinx (2021b). Zynq-7000 SoC Technical Reference Manual (UG585). Available online at: www.xilinx.com

AMD Xilinx (2023a). Versal Adaptive SoC Technical Reference Manual (AM011) Available online at: www.xilinx.com

AMD Xilinx (2023b). Versal HBM Series (Product Brief) Available online at: www.xilinx.com

AMD Xilinx (2023c). Zynq UltraScale+ Device Technical Reference Manual (UG1085). Available online at: www.xilinx.com

AMD Xilinx (2023d). Zynq UltraScale+ MPSoC Embedded Design Methodology Guide (UG1228). Available online at: www.xilinx.com

AMD Xilinx (2024). Versal Architecture and Product Data Sheet: Overview (AMD Xilinx Datasheet: DS950 v1.21) Available online at: www.xilinx.com

Arm Limited (2021). AMBA AXI and ACE Protocol Specification. Available online at: www.arm.com

Association for Computing Machinery (2016). Artifact Review and Badging. Available online at: https://www.acm.org/publications/policies/artifact-review-badging Accessed: March 14, 2018

Axer, M. and Amunts, K. (2022). Scale matters: The nested human connectome. *Science* 378, 500–504. doi:10.1126/science.abq2599

Barba, L. A. (2018). Terminologies for Reproducible Research arXiv:1802.03311

Benureau, F. C. Y. and Rougier, N. P. (2017). Re-run, Repeat, Reproduce, Reuse, Replicate: Transforming Code into Scientific Contributions. *CoRR* abs/1708.08205

Bhattacharjee, K. and Das, S. (2022). A search for good pseudo-random number generators: Survey and empirical studies. *Computer Science Review* 45, 100471. doi:https://doi.org/10.1016/j.cosrev.2022.100471

Blundell, I., Brette, R., Cleland, T. A., Close, T. G., Coca, D., Davison, A. P., et al. (2018a). Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics* 12. doi:10.3389/fninf.2018.00068

Blundell, I., Brette, R., Cleland, T. A., Close, T. G., Coca, D., Davison, A. P., et al. (2018b). Code Generation in Computational Neuroscience: A Review of Tools and Techniques. *Frontiers in Neuroinformatics* 12, 68. doi:10.3389/fninf.2018.00068

Blundell, I., Plotnikov, D., Eppler, J. M., and Morrison, A. (2018c). Automatically selecting an optimal integration scheme for systems of differential equations in neuron models. *Frontiers in Neuroinformatics*

Bourque, P. and Fairley, R. E. (eds.) (2014). SWEBOK: Guide to the Software Engineering Body of Knowledge (Los Alamitos, CA: IEEE Computer Society), version 3.0 edn.

Braitenberg, V. and Schüz, A. (1998). Cortex: Statistics and Geometry of Neuronal Connectivity (Berlin, Heidelberg: Springer Berlin Heidelberg). doi:10.1007/978-3-662-03733-1

Brunel, N. (2000). Dynamics of Sparsely Connected Networks of Excitatory and Inhibitory Spiking Neurons. *Journal of Computational Neuroscience* 8. doi:10.1023/A:1008925309027

Brunel, N. and Van Rossum, M. C. W. (2007a). Lapicque's 1907 paper: from frogs to integrate-and-fire. *Biological Cybernetics* 97, 337–339. doi:10.1007/s00422-007-0190-0

Brunel, N. and Van Rossum, M. C. W. (2007b). Quantitative investigations of electrical nerve excitation treated as polarization: Louis Lapicque 1907 · Translated by:. *Biological Cybernetics* 97, 341–349. doi:10.1007/s00422-007-0189-6

Cheung, K., Schultz, S. R., and Luk, W. (2016). NeuroFlow: A General Purpose Spiking Neural Network Simulation Platform using Customizable Processors. *Frontiers in Neuroscience* 9. doi:10.3389/fnins.2015.00516

Cohen, J. (1988). Statistical Power Analysis for the Behavioral Sciences (Lawrence Erlbaum Associates)

Cornell Aeronautical Laboratory Inc. (1960). Mark I Perceptron operatores' manual. Available online at: https://apps.dtic.mil/sti/tr/pdf/AD0236965.pdf Accessed: November 16, 2023

Dahmen, W. and Reusken, A. (2005). Numerik für Naturwissenschaftler (Springer)

Dasbach, S., Tetzlaff, T., Diesmann, M., and Senk, J. (2021). Dynamical Characteristics of Recurrent Neuronal Networks Are Robust Against Low Synaptic Weight Resolution. *Frontiers in Neuroscience* 15, 757790. doi:10.3389/fnins.2021.757790

Davies, M., Srinivasa, N., Lin, T., Chinya, G., Cao, Y., Choday, S. H., et al. (2018). Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro* 38, 82–99. doi: 10.1109/MM.2018.112130359

Davison, A., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., et al. (2009). PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics* 2, 11. doi:10.3389/neuro.11.011.2008

Dayan, P., Abbott, L. F., and Abbott, L. F. (2005). Theoretical neuroscience: computational and mathematical modeling of neural systems. Computational neuroscience (Cambridge, Mass.: MIT Press), first paperback ed edn.

Eliasmith, C. (2013). How to build a brain: A neural architecture for biological cognition (Oxford: Oxford University Press)

Eppler, J., Helias, M., Muller, E., Diesmann, M., and Gewaltig, M.-O. (2009). PyNEST: a convenient interface to the NEST simulator. *Frontiers in Neuroinformatics* 2. doi:10.3389/neuro.11.012.2008

Fardet, T., Vennemo, S. B., Mitchell, J., Mørk, H., Graber, S., Hahne, J., et al. (2020). NEST 2.20.1. doi:10.5281/zenodo.4018718

FitzHugh, R. (1961). Impulses and Physiological States in Theoretical Models of Nerve Membrane. *Biophysical Journal* 1, 445–466. doi:10.1016/S0006-3495(61)86902-6

Friedmann, S., Schemmel, J., Grübl, A., Hartel, A., Hock, M., and Meier, K. (2017). Demonstrating Hybrid Learning in a Flexible Neuromorphic Hardware System. *IEEE Transactions on Biomedical Circuits and Systems* 11, 128–142. doi:10.1109/TBCAS.2016.2579164

Furber, S. and Bogdan, P. (2020). Spinnaker - A Spiking Neural Network Architecture (NOW Publishers INC). OCLC: 1137208476

Furber, S. B., Lester, D. R., Plana, L. A., Garside, J. D., Painkras, E., Temple, S., et al. (2013). Overview of the SpiNNaker System Architecture. *IEEE Transactions on Computers* 62, 2454–2467. doi:10.1109/TC.2012.142

George, M. and Alfke, P. (2007). Linear Feedback Shift Registers in Virtex Devices (AMD Xilinx Application Note: XAPP 052) Available online at: www.xilinx.com

Gerstner, W., Kistler, W. M., Naud, R., and Paninski, L. (2014). Neuronal dynamics: from single neurons to networks and models of cognition (Cambridge, United Kingdom: Cambridge University Press)

Gewaltig, M.-O. and Diesmann, M. (2007). NEST (NEural Simulation Tool). *Scholarpedia* 2, 1430

Gleeson, P., Crook, S., Cannon, R. C., Hines, M. L., Billings, G. O., Farinella, M., et al. (2010). NeuroML: a language for describing data driven models of neurons and networks with a high degree of biological detail. *PLoS Comput. Biol.* 6, e1000815

Golosio, B., Tiddia, G., De Luca, C., Pastorelli, E., Simula, F., and Paolucci, P. S. (2021). Fast Simulations of Highly-Connected Spiking Cortical Models Using GPUs. *Frontiers in Computational Neuroscience* 15. doi:10.3389/fncom.2021.627620

Goodman, D. and Brette, R. (2008). Brian: a simulator for spiking neural networks in python. *Frontiers in neuroinformatics* 2, 5. doi:10.3389/neuro.11.005.2008

Goodman, S. N., Fanelli, D., and Ioannidis, J. P. A. (2016). What does research reproducibility mean? *Science Translational Medicine* 8, 341ps12–341ps12. doi:10.1126/scitranslmed.aaf5027

Grün, S. and Rotter, S. (eds.) (2010). Analysis of Parallel Spike Trains (Boston, MA: Springer US). doi:10.1007/978-1-4419-5675-0

Gutzen, R., von Papen, M., Trensch, G., Quaglio, P., Grün, S., and Denker, M. (2018). Reproducible Neural Network Simulations: Statistical Methods for Model Validation on the Level of Network Activity Data. *Frontiers in Neuroinformatics* 12, 90. doi:10.3389/fninf.2018.00090

Hansel, D., Mato, G., Meunier, C., and Neltner, L. (1998). On Numerical Simulations of Integrate-and-Fire Neural Networks. *Neural Computation* 10, 467–483. doi:10.1162/089976698300017845

Hebb, D. O. (1949). The Organization of Behavior (New York: Wiley)

Heittmann, A., Psychou, G., Trensch, G., Cox, C. E., Wilcke, W. W., Diesmann, M., et al. (2022). Simulating the Cortical Microcircuit Significantly Faster Than Real Time on the IBM INC-3000 Neural Supercomputer. *Frontiers in Neuroscience* doi:10.3389/fnins.2021.728460

Hines, M. L. and Carnevale, N. T. (1997). The NEURON simulation environment. *Neural computation*

Hines, M. L. and Carnevale, N. T. (2000). Expanding NEURON's repertoire of mechanisms with NMODL. *Neural computation* 12, 995–1007. doi:10.1162/089976600300015475

Hodgkin, A. L. and Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology* 117, 500–544. doi:10.1113/jphysiol.1952.sp004764

Hopkins, M. and Furber, S. (2015). Accuracy and Efficiency in Fixed-Point Neural ODE Solvers. *Neural Computation* 27, 2148–2182. doi:10.1162/NECO_a_00772. PMID: 26313605

Hubel, D. H. and Wiesel, T. N. (2005). Brain and visual perception: the story of a 25-year collaboration (New York, N.Y: Oxford University Press)

Höppner, S., Yan, Y., Dixius, A., Scholze, S., Partzsch, J., Stolba, M., et al. (2021). The SpiNNaker 2 Processing Element Architecture for Hybrid Digital Neuromorphic Computing

Indiveri, G., Linares-Barranco, B., Hamilton, T. J., Schaik, A. v., Etienne-Cummings, R., Delbruck, T., et al. (2011). Neuromorphic Silicon Neuron Circuits. *Frontiers in Neuroscience* 5, 73. doi:10.3389/fnins.2011.00073

Izhikevich, E. M. (2003). Simple Model of Spiking Neurons. *Trans. Neur. Netw.* 14, 1569–1572. doi:10.1109/TNN.2003.820440

Izhikevich, E. M. (2006). Polychronization: Computation with Spikes. *Neural Computation* 18, 245–282

Kandel, E. R. (2007). In search of memory: the emergence of a new science of mind (New York: Norton)

Kandel, E. R., Schwartz, J. H., and Jessell, T. M. (eds.) (2000). *Principles of Neural Science* (New York: McGraw-Hill, Health Professions Division), chap. 2. 4th edn., 19–35

Kauth, K., Stadtmann, T., Brandhofer, R., Sobhani, V., and Gemmeke, T. (2020). Communication Architecture Enabling 100x Accelerated Simulation of Biological Neural Networks. In *Proceedings of the Workshop on System-Level Interconnect: Problems and Pathfinding Workshop* (New York, NY, USA: Association for Computing Machinery), SLIP '20. doi:10.1145/3414622.3431909

Kauth, K., Stadtmann, T., Sobhani, V., and Gemmeke, T. (2023). neuroAIx-Framework: design of future neuroscience simulation systems exhibiting execution of the cortical microcircuit

model 20× faster than biological real-time. *Frontiers in Computational Neuroscience* 17. doi:10.3389/fncom.2023.1144143

Knight, J. C. and Nowotny, T. (2018). GPUs Outperform Current HPC and Neuromorphic Solutions in Terms of Speed and Energy When Simulating a Highly-Connected Cortical Model. *Frontiers in Neuroscience* 12. doi:10.3389/fnins.2018.00941

Knight, J. C. and Nowotny, T. (2021). Larger GPU-accelerated brain simulations with procedural connectivity. *Nature Computational Science* 1, 136–142. doi:10.1038/s43588-020-00022-7

Kunkel, S., Schmidt, M., Eppler, J. M., Plesser, H. E., Masumoto, G., Igarashi, J., et al. (2014). Spiking network simulation code for petascale computers. *Frontiers in Neuroinformatics* 8. doi:10.3389/fninf.2014.00078

Kurth, A. C., Senk, J., Terhorst, D., Finnerty, J., and Diesmann, M. (2022). Sub-realtime simulation of a neuronal network of natural density. *Neuromorphic Computing and Engineering* 2, 021001. doi:10.1088/2634-4386/ac55fc

Kusch, L., Diaz, S., Klijn, W., Sontheimer, K., Bernard, C., Morrison, A., et al. (2022). Multiscale cosimulation design template implemented for a neuroscience application. doi:10.1101/2022.07.13.499940

Lambert, J. D. (1992). Numerical Methods for Ordinary Differential Systems (Wiley)

Lapicque, L. (1907). Recherches quantitatives sur l'excitation électrique des nerfs traitée comme une polarisation. *J. Physiol. Pathol. Gen. 9* , 620–635

Maguire, L., McGinnity, T., Glackin, B., Ghani, A., Belatreche, A., and Harkin, J. (2007). Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing* 71, 13–29. doi:10.1016/j.neucom.2006.11.029

Mahowald, M. (1992). *VLSI Analogs of Neuronal Visual Processing: A Synthesis of Form and Function*. Ph.D. thesis, Califpionia In stitute of Technology, Pasadella, California

Marsaglia, G. (1995). The Marsaglia Random Number CDROM including the Diehard Battery of Tests of Randomness. Available online at: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/ Archived from https://ani.stat.fsu.edu/diehard/ on 2016-01-25

Martin, R. C. and Coplien, J. O. (2009). Clean code: a handbook of agile software craftsmanship (Prentice Hall)

Mead, C. (1989). Analog VLSI and Neural Systems. Computation and neural systems series (Reading, Mass: Addison-Wesley)

Mead, C. (1990). Neuromorphic electronic systems. In *Proc. IEEE, 78:16291636*

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., et al. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science* 345, 668–673

Micron (2006). DDR3 SDRAM Datasheet MT41J256M8. Available online at: www.micron.com.

Modha, D. S., Akopyan, F., Andreopoulos, A., Appuswamy, R., Arthur, J. V., Cassidy, A. S., et al. (2023a). IBM NorthPole Neural Inference Machine. In *2023 IEEE Hot Chips 35 Symposium (HCS)*. 1–58. doi:10.1109/HCS59251.2023.10254721

Modha, D. S., Akopyan, F., Andreopoulos, A., Appuswamy, R., Arthur, J. V., Cassidy, A. S., et al. (2023b). Neural inference at the frontier of energy, space, and time. *Science* 382, 329–335. doi:10.1126/science.adh1174

Moore, S. W., Fox, P. J., Marsh, S. J., Markettos, A. T., and Mujumdar, A. (2012). Bluehive - A field-programable custom computing machine for extreme-scale real-time neural network simulation. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 133–140. doi:10.1109/FCCM.2012.32

Morrison, A., Diesmann, M., and Gerstner, W. (2008). Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics* 98, 459–478. doi:10.1007/s00422-008-0233-1

Morrison, A., Mehring, C., Geisel, T., Aertsen, A., and Diesmann, M. (2005). Advancing the Boundaries of High-Connectivity Network Simulation with Distributed Computing. *Neural Computation* 17, 1776–1801. doi:10.1162/0899766054026648

Morrison, A., Straube, S., Plesser, H. E., and Diesmann, M. (2007). Exact subthreshold integration with continuous spike times in discrete time neural network simulations. *Neural Computation* 19, 47–79

Nagumo, J., Arimoto, S., and Yoshizawa, S. (1962). An Active Pulse Transmission Line Simulating Nerve Axon. *Proceedings of the IRE* 50, 2061–2070. doi:10.1109/JRPROC.1962.288235

Narayanan, P., Cox, C. E., Asseman, A., Antoine, N., Huels, H., Wilcke, W. W., et al. (2020). Overview of the IBM Neural Computer Architecture. doi:10.48550/ARXIV.2003.11178

Noll, T. G., von Sydow, T., Neumann, B., Schleifer, J., Coenen, T., and Kappen, G. (2010). Reconfigurable Components for Application-Specific Processor Architectures. In *Dynamically Reconfigurable Systems*, eds. M. Platzner, J. Teich, and N. Wehn (Heidelberg: Springer), chap. 2. 25–49. doi:DOI10.1007/978-90-481-3485-4

Nordlie, E., Gewaltig, M.-O., and Plesser, H. E. (2009). Towards Reproducible Descriptions of Neuronal Network Models. *PLoS Computational Biology* 5, e1000456. doi:10.1371/journal.pcbi.1000456

Pani, D., Meloni, P., Tuveri, G., Palumbo, F., Massobrio, P., and Raffo, L. (2017). An FPGA Platform for Real-Time Simulation of Spiking Neuronal Networks. *Frontiers in Neuroscience* 11. doi:10.3389/fnins.2017.00090

Patil, P., Peng, R. D., and Leek, J. (2016). A statistical definition for reproducibility and replicability. *bioRxiv* doi:10.1101/066803

Pauli, R., Weidel, P., Kunkel, S., and Morrison, A. (2018). Reproducing Polychronization: A Guide to Maximizing the Reproducibility of Spiking Network Models. *Frontiers in Neuroinformatics* 12. doi:10.3389/fninf.2018.00046

Pehle, C., Billaudelle, S., Cramer, B., Kaiser, J., Schreiber, K., Stradmann, Y., et al. (2022). The BrainScaleS-2 Accelerated Neuromorphic System With Hybrid Plasticity. *Frontiers in Neuroscience* 16. doi:10.3389/fnins.2022.795876

Pfeil, T., Potjans, T., Schrader, S., Potjans, W., Schemmel, J., Diesmann, M., et al. (2012). Is a 4-Bit Synaptic Weight Resolution Enough? – Constraints on Enabling Spike-Timing Dependent Plasticity in Neuromorphic Hardware. *Frontiers in Neuroscience* 6. doi:10.3389/fnins.2012.00090

Plesser, H. E. (2018). Reproducibility vs. Replicability: A Brief History of a Confused Terminology. *Frontiers in Neuroinformatics*

Plotnikov, D., Blundell, I., Ippen, T., Eppler, J. M., Morrison, A., and Rumpe, B. (2016). NESTML: a modeling language for spiking neurons. In *Modellierung 2016, 2.-4. März 2016, Karlsruhe*. 93–108

Potjans, T. C. and Diesmann, M. (2014). The Cell-Type Specific Cortical Microcircuit: Relating Structure and Activity in a Full-Scale Spiking Network Model. *Cerebral Cortex* 24, 785–806. doi:10.1093/cercor/bhs358

Pronold, J., Jordan, J., Wylie, B. J. N., Kitayama, I., Diesmann, M., and Kunkel, S. (2022). Routing Brain Traffic Through the Von Neumann Bottleneck: Parallel Sorting and Refactoring. *Frontiers in Neuroinformatics* 15. doi:10.3389/fninf.2021.785068

Rotter, S. and Diesmann, M. (1999). Exact digital simulation of time-invariant linear systems with applications to neuronal modeling. *Biological Cybernetics* 81, 381–402

Rowley, A., Stokes, A., and Gait, A. (2017). SpiNNaker new model template lab manual

Savio Martis, M. (2006). Validation of simulation based models: A theoretical outlook 4

Schemmel, J., Brüderle, D., Grübl, A., Hock, M., Meier, K., and Millner, S. (2010). A Wafer-Scale Neuromorphic Hardware System for Large-Scale Neural Modeling. *Proceedings of the 2010 IEEE International Symposium on Circuitsand Systems (ISCAS"10)* , 1947–1950

Schemmel, J., Kriener, L., Muller, P., and Meier, K. (2017). An Accelerated Analog Neuromorphic Hardware System Emulating NMDA- and Calcium-Based Non-Linear Dendrites. In *2017 International Joint Conference on Neural Networks (IJCNN)* (Anchorage, AK, USA: IEEE), 2217–2226. doi:10.1109/IJCNN.2017.7966124

Schlesinger, S., Crosbie, R. E., Gagné, R. E., Innes, G. S., Lalwani, C., Loch, J., et al. (1979). Terminology for Model Credibility. *Simulation* 32, 103–104

Schmidt, M., Bakker, R., Hilgetag, C. C., Diesmann, M., and Van Albada, S. J. (2018a). Multi-scale account of the network structure of macaque visual cortex. *Brain Structure and Function* 223, 1409–1435. doi:10.1007/s00429-017-1554-4

Schmidt, M., Bakker, R., Shen, K., Bezgin, G., Diesmann, M., and Van Albada, S. J. (2018b). A multi-scale layer-resolved spiking network model of resting-state dynamics in macaque visual cortical areas. *PLOS Computational Biology* 14, e1006359. doi:10.1371/journal.pcbi.1006359

Schuecker, J., Schmidt, M., Van Albada, S. J., Diesmann, M., and Helias, M. (2017). Fundamental Activity Constraints Lead to Specific Interpretations of the Connectome. *PLOS Computational Biology* 13, e1005179. doi:10.1371/journal.pcbi.1005179

Schuman, C. D., Kulkarni, S. R., Parsa, M., Mitchell, J. P., Date, P., and Kay, B. (2022). Opportunities for neuromorphic computing algorithms and applications. *Nature Computational Science* 2, 10–19. doi:10.1038/s43588-021-00184-y

Schuman, C. D., Potok, T. E., Patton, R. M., Birdwell, J. D., Dean, M. E., Rose, G. S., et al. (2017). A Survey of Neuromorphic Computing and Neural Networks in Hardware. doi:10.48550/arXiv.1705.06963. ArXiv:1705.06963 [cs]

Senk, J., Kriener, B., Djurfeldt, M., Voges, N., Jiang, H.-J., Schüttler, L., et al. (2022). Connectivity concepts in neuronal network modeling. *PLOS Computational Biology* 18, e1010086. doi:10.1371/journal.pcbi.1010086

Senk, J., Yegenoglu, A., Amblet, O., Brukau, Y., Davison, A., Lester, D. R., et al. (2017). A Collaborative Simulation-Analysis Workflow for Computational Neuroscience Using HPC. In *High-Performance Scientific Computing / Di Napoli, Edoardo (Editor) ; Cham : Springer International Publishing, 2017, Chapter 21 ; ISSN: 0302-9743=1611-3349 ; ISBN: 978-3-319-53861-7=978-3-319-53862-4 ; doi:10.1007/978-3-319-53862-4*. Jülich Aachen Research Alliance (JARA) High-Performance Computing Symposium, Aachen (Germany), 4 Oct 2016 - 5 Oct 2016 (Cham: Springer International Publishing), vol. 10164 of *Lecture Notes in Computer Science*, 243 – 256. doi:10.1007/978-3-319-53862-4_21

Shinomoto, S., Shima, K., and Tanji, J. (2003). Differences in Spiking Patterns Among Cortical Neurons. *Neural Computation* 15, 2823–2842. doi:10.1162/089976603322518759

Sommerville, I. (2015). Software Engineering (Pearson Education), 10th edn.

Spreizer, S., Senk, J., Rotter, S., Diesmann, M., and Weyers, B. (2021). Nest desktop, an educational application for neuroscience. *eNeuro* 8. doi:10.1523/ENEURO.0274-21.2021

Stokes, A., Rowley, A., Brenninkmeijer, C., Fellows, D., Rhodes, O., Gait, A., et al. (2007). SpiNNaker software stack

Strehmel, K. and Weiner, R. (1995). Numerik gewöhnlicher Differentialgleichungen (B.G. Teubner)

Suarez, E., Eicker, N., and Lippert, T. (2019). *Modular Supercomputing Architecture: from Idea to Production; 3rd* (FL, USA: CRC Press), vol. 3. 223–251

Suarez, E., Kunkel, S., Küsters, A., Plesser, H. E., and Lippert, T. (2021). Modular supercomputing for neuroscience. In *Brain-Inspired Computing*, eds. K. Amunts, L. Grandinetti, T. Lippert, and N. Petkov (Cham: Springer International Publishing), 63–80

Tappert, C. C. (2019). Who is the father of deep learning? In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*. 343–348. doi:10.1109/CSCI49370.2019.00067

Tausworthe, R. C. (1965). Random numbers generated by linear recurrence modulo two. *Mathematics of Computation* 19, 201–209. doi:10.1090/S0025-5718-1965-0184406-1

Temple, S. (2011a). AppNote 1 - SpiNN-3 Development Board. Available online at: http://spinnakermanchester.github.io/docs/spinn-app-1.pdf Accessed: March 14, 2018

Temple, S. (2011b). AppNote 4 - SpiNNaker Datagram Protocol (SDP) Specification. Available online at: http://spinnakermanchester.github.io/docs/spinn-app-4.pdf Accessed: March 14, 2018

Thacker, B., Doebling, S., Hemez, F., Anderson, M., Pepin, J., and Rodriguez, E. (2004). Concepts of Model Verification and Validation doi:10.2172/835920

Trensch, G., Gutzen, R., Blundell, I., Denker, M., and Morrison, A. (2018). Rigorous Neural Network Simulations: A Model Substantiation Methodology for Increasing the Correctness of Simulation Results in the Absence of Experimental Validation Data. *Frontiers in Neuroinformatics* 12, 81. doi:10.3389/fninf.2018.00081

van Albada, S. J., Rowley, A. G., Senk, J., Hopkins, M., Schmidt, M., Stokes, A. B., et al. (2018). Performance Comparison of the Digital Neuromorphic Hardware SpiNNaker and the Neural Network Simulation Software NEST for a Full-Scale Cortical Microcircuit Model. *Frontiers in Neuroscience* 12. doi:10.3389/fnins.2018.00291

Wang, R., Hamilton, T. J., Tapson, J., and van Schaik, A. (2014). An FPGA design framework for large-scale spiking neural networks. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*. 457–460. doi:10.1109/ISCAS.2014.6865169

Wang, R. M., Thakur, C. S., and van Schaik, A. (2018). An FPGA-Based Massively Parallel Neuromorphic Cortex Simulator. *Frontiers in Neuroscience* 12, 213. doi:10.3389/fnins.2018.00213

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 57
Plasma Breakdown and Runaway Modelling in ITER-scale Tokamaks
J. Chew (2023), xv, 172 pp
ISBN: 978-3-95806-730-1

Band / Volume 58
Space Usage and Waiting Pedestrians at Train Station Platforms
M. Küpper (2023), ix, 95 pp
ISBN: 978-3-95806-733-2

Band / Volume 59
Quantum annealing and its variants: Application to quadratic unconstrained binary optimization
V. Mehta (2024), iii, 152 pp
ISBN: 978-3-95806-755-4

Band / Volume 60
Elements for modeling pedestrian movement from theory to application and back
M. Chraibi (2024), vi, 279 pp
ISBN: 978-3-95806-757-8

Band / Volume 61
Artificial Intelligence Framework for Video Analytics:
Detecting Pushing in Crowds
A. Alia (2024), xviii, 151 pp
ISBN: 978-3-95806-763-9

Band / Volume 62
The Relationship between Pedestrian Density, Walking Speed and Psychological Stress:
Examining Physiological Arousal in Crowded Situations
M. Beermann (2024), xi, 117 pp
ISBN: 978-3-95806-764-6

Band / Volume 63
Eventify Meets Heterogeneity:
Enabling Fine-Grained Task-Parallelism on GPUs
L. Morgenstern (2024), xv, 110 pp
ISBN: 978-3-95806-765-3

Band / Volume 64
Dynamic Motivation in Crowds: Insights from Experiments and Pedestrian Models for Goal-Directed Motion
E. Üsten (2024), ix, 121 pp
ISBN: 978-3-95806-773-8

Schriften des Forschungszentrums Jülich
IAS Series

Band / Volume 65
Propagation of Stimuli in Crowds:
Empirical insights into mutual influence in human crowds
H. Lügering (2024), xi, 123 pp
ISBN: 978-3-95806-775-2

Band / Volume 66
Classification of Pedestrian Streams: From Empirics to Modelling
J. Cordes (2024), vii, 176 pp
ISBN: 978-3-95806-780-6

Band / Volume 67
Optimizing Automated Shading Systems in Office Buildings by Exploring Occupant Behaviour
G. Derbas (2024), 9, x, 168, ccxxiii
ISBN: 978-3-95806-787-5

Band / Volume 68
Speed-Density Analysis in Pedestrian Single-File Experiments
S. Paetzke (2025), XIII, 107 pp
ISBN: 978-3-95806-818-6

Band / Volume 69
Proceedings of the 35th Parallel Computational Fluid Dynamics International Conference 2024
A. Lintermann, S. S. Herff, J. H. Göbbert (2025), xv, 321 pp
ISBN: 978-3-95806-819-3

Band / Volume 70
Towards Improved Civil Safety: Experimental Insights into Impulse Propagation through Crowds
S. Feldmann (2025), xi, 99 pp
ISBN: 978-3-95806-828-5

Band / Volume 71
Development and Evaluation of Architecture Concepts for a System-on-Chip Based Neuromorphic Compute Node for Accelerated and Reproducible Simulations of Spiking Neural Networks in Neuroscience
G. Trensch (2025), 219, XXXV pp
ISBN: 978-3-95806-832-2

Weitere ***Schriften des Verlags im Forschungszentrum Jülich*** unter
http://wwwzb1.fz-juelich.de/verlagextern1/index.asp

IAS Series
Band / Volume 71
ISBN 978-3-95806-832-2

Mitglied der Helmholtz-Gemeinschaft