CONFIGURING A MASSIVELY PARALLEL CMP SYSTEM FOR REAL-TIME NEURAL APPLICATIONS

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2009

By
M. Mukaram Khan
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

Configuring a million-core parallel system at boot time is a difficult process when the system has neither specialised boot-up hardware support nor a preconfigured default state that puts it in operating condition. The SpiNNaker massively-parallel computing system has been designed to support large-scale simulations of biologically-inspired neural networks in real-time. The system building block is a Chip Multiprocessor (CMP) using low-power embedded processors, with an asynchronous network-on-chip to support high-performance, scalable, and fault-tolerant parallel distributed processing. Where most large CMP systems feature a sideband network to complete the boot process, SpiNNaker has a single homogeneous network interconnect for both application inter-processor communication and system control functions such as boot load and run-time user-system interaction. This network improves fault tolerance and makes it easier to support dynamic run-time reconfiguration. However, it requires a boot process that is transaction-level compatible with the application's communications model.

Since SpiNNaker uses event-driven asynchronous communication throughout, the loader operates with purely local control: there is no global synchronisation, state information, or transition sequence. A novel two-stage unfolding boot-up process efficiently configures a multi-CMP SpiNNaker into an integrated computing system and loads the application using a high-speed flood-fill technique with support for run-time reconfiguration. SystemC simulation of a multi-CMP SpiNNaker system indicates an error-free CMP configuration time of ≈1.3 ms, while a high-level simulation of a full-scale system (with 64,000 CMPs) indicates a mean application-loading time of ≈20 ms for a 100-Kbyte application that is virtually independent of the size of the system.

The configuration process also supports application development through a hardware abstraction layer (HAL) that provides architectural visibility appropriate to the developer's purpose.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the "Copyright") and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of School of Computer Science (or the Vice-President).

Acknowledgements

I would like to thank my colleagues in the APT group, especially from the SpiNNaker project, for providing a wonderful company without which IT302 would have been a boring place. I want to thank them all for their support that saved me a lot of effort and time. My special thanks to Viv Woods for refining my writings which let me concentrate on my work. Luis has always been available for help when ever required. The list of my publication would have not been that long if Alex was not on our side, and my implementation would have been behind schedule if Xin Jin was not there to help me. Steve Temple, Jim Garside and Mikel have been very patient in helping me refine my protocols and their verification. Eustice, Jian, Shufan, Yebin, Dom, Cameron, Frencesco, and Zenyu have been a great help on many occasions in enabling me to understand part of my work related to their research areas. Javier's helpful and timely response from Spain made a long-distance collaboration possible during my research. I wish to thank them all for their time and help. I also want to thank our partners in the University of Southampton for their hospitality and nice company during this project.

My special thanks to Steve Furber for being so helpful to me. He has always been an inspiration for me during this research, and I look forward to having a continued research collaboration with him during my academic career.

I dedicate my thesis to my wife Sabeen and children Muazam, Maham and Mahad, who deserved much more time and attention than I could manage to give them during my PhD.

My thanks to E. Izhikevich and P. Dayan for permitting me to use diagrams from their publications. Last but not the least, I am grateful to NUST Pakistan, HEC Pakistan, and EPSRC UK who supported me in paying my utility bills.

Chapter 1

Introduction

"As our understanding grows about how the brain perceives, thinks and remembers, so does our ability to devise machine intelligence ..... When it was realized that the brain is not a serial processor of information but rather an extraordinarily complex parallel system, the field of computer science turned to a new kind of computer - massively parallel processors - .... The long-term possibilities have the flavour of science fiction and raise the most basic questions of philosophy and ethics. Can we someday create machine minds as capable as, or even more capable than, the human mind in all its ramifications, including the ability to create and evaluate new concepts? .. Can we someday develop instrumentation to "read" the mind? Can thoughts and knowledge someday be implanted in the mind or transferred from one mind to another? Can our intellectual capabilities someday be substantially enhanced by "symbiosis" with machine intelligence? .....The pace of developments in neuroscience and in computer science is so rapid and accelerating that many of these science-fiction-like possibilities may become realities in your lifetime." Richard F. Thompson [Tho00]

1.1 Background

By most qualified definitions, a computer is taken to be "a machine that manipulates data according to a list of instructions" [Wik09b]. In more general terms, "a computer is an electronic machine that can receive, process and present information [based on a set of instructions]" [Wik09c]. Originally the term referred to a person who did mathematical calculations using a mechanical device [Wik09b]. From their inception, it was commonplace to compare computers with brains and

to find similarities between the two. However, the two "devices" outweigh each other in their respective strengths: computers are fast, accurate and have a large capacity to store data, while brains are flexible, fault-tolerant and highly concurrent in their operations [FT07]. There have been many endeavours to design a computing device with a union of the two differing domains, though the two contribute to the same end. Over the last half a century or more of the evolution of programmable computers, we (computer scientists) have been trying to bridge this gap and are learning from neurobiology to build "brain like computers": computing devices characterised by their ability to adapt to the environment, having an interactive and error-prone behaviour, employing highly-parallel distributed processing, and still being fast and accurate. However, this requires a detailed understanding of the brain which we are still lacking.

The brain is probably the most complex structure in the universe and is the last mystery to be revealed by humans [Tra02]. Our brain consists of 10^{11} neurons having more than 10^{15} connections each of which can transmit an information signal a few times a second. Each neuron, with its connections, contributes to a very complex biological system which has not yet been fully understood. The least understood aspect is the emergent behaviour exhibited by a population of neurons (a neural network), which collectively display a holistic behaviour quite different from individual neurons. Many discoveries over the last few decades have improved our understanding of the nervous system. New techniques, such as Functional Magnetic Resonance Imaging (fMRI) etc, are adding considerably to this knowledge-base. Mathematical models based on these discoveries and computer simulations based on hypothesised mathematical models are helping us to understand phenomenological observations of neurons and their interactions. A few detailed mathematical models can simulate biological neurons realistically, however, they are too complex to simulate a large population of neurons on a general-purpose computer [FT07].

The SpiNNaker project at the University of Manchester aims to realise large-scale neural simulations using biologically realistic mathematical models. Our aim is two-fold: firstly adopting known biological engineering principles from the brain to develop a fault-tolerant massively-parallel computing system using a low-power architecture. Secondly, using this large-scale neural network simulation engine in collaboration with multi-disciplinary researchers to discover more about the brain's functionality. It is intended to use the resulting discoveries to improve

the hardware system, making it operate in a manner more akin to the human brain [FT07]. This iterative process to "explore and synthesize", in order to bridge the gap between the man-made and natural computing systems, should help us learn how to build a more flexible, interactive and robust computing device.

SpiNNaker is based upon an Application Specific Integrated Circuit (ASIC) design using state-of-the-art technologies such as the Chip Multiprocessor (CMP) and asynchronous inter-processor communication. To simulate a large-population of neurons, it uses an event-driven application model akin to real-time embedded applications. The architecture and application model are quite different from those used in conventional computing systems. For an application to make optimal use of the designed features, it is important to configure the SpiNNaker system properly and manage its resources at run-time to support real-time simulations of large-scale neural networks.

1.2 Motivation

The main motivation for this research has spun out from the objectives of designing the SpiNNaker massively-parallel computing system, i.e. to support understanding the brain by providing a high-performance simulation engine to support large-scale neural modelling to the scale of a part of the nervous system. This requires researchers from multiple disciplines to use SpiNNaker for a variety of applications using various kinds of mathematical models (spiking neurons vs. multi-layer perceptrons, biologically realistic neurons vs. simple spiking neurons etc). It implies that developers would wish to develop their applications for SpiNNaker without delving into its architectural complexities, or to port their existing applications to the SpiNNaker computing system without much difficulty. This requires configuration of the hardware for specific application requirements, providing support to the developers in the form of ready-made functions to hide the architectural details at higher levels of implementation, providing middle-layer support functions to help transform an application into the SpiNNaker-application-model, and mapping the neural network to the available resources to make optimal use of them.

In this research we aim to provide application support for real-time large-scale neural simulations on the SpiNNaker multi-CMP system. We intend to provide a

personal-computer (PC) like environment to application developers, application users and system administrators interacting with SpiNNaker. We want to motivate developers to develop or to port existing applications to SpiNNaker with minimal effort. We also wish to support an interactive system administration for resource management and run-time fault handling.

1.3 Aim and Objectives

The aim of this research is to "devise a configuration process for a multi-CMP system to configure the system at run-time as required by the application, and then to load the tailored application at run-time using the SpiNNaker inter-chip communication interconnect". The research is directed toward achieving the following objectives:

- Performance: To make the system available to the application by efficiently testing and initializing the SpiNNaker-CMPs at configuration time, and interactively configuring the system, as a whole, at run-time as required by the application. This also implies that the system be configured to make optimal use of its resources, and support the application execution to make full use of its designed features.

- Scalability: SpiNNaker is a scalable computing system i.e. a system of almost any desired scale can be assembled by linking the SpiNNaker CMPs together. The configuration process should be independent of the scale of the system with regards to initialization and loading applications.

- Fault-tolerance: SpiNNaker has been designed as a reconfigurable and fault-tolerant computing system. However, the system depends on the software to make full use of these features. The configuration process should use these features to support autonomous fault recovery at run-time. At the same time, the configuration process itself should be robust enough to be able to bring the system to an operational state despite some chip- or system-level malfunctions.

- Interactive Support: The process should provide run-time support to enable users to interact with the application and a visualization of the state of the application or the system at any point in time. This is important, for

example, to provide stimuli to the neural application at run time and to get responses as a result of these stimuli. It is also important for system diagnostic and debugging support.

1.4 Contributions

The research has contributed in providing:

- A high-level instruction- and cycle-accurate simulation model of a multi-CMP system, developed much before the hardware design of the systems to support the conceptual validation of the application model, detailed design verification and architectural exploration, and application development for the SpiNNaker multi-CMP system (Chapter 5).
- A novel process to configure a multi-CMP system at run-time using a real-time event-driven application model, with characteristics of efficiency, scalability, and fault-tolerance (Chapter 7).
- A novel event-driven application loading process to load a tailored application efficiently, from outside the system, onto a multi-CMP system at run-time in a scalable way (Chapter 7).
- A novel real-time chip- and system-level fault-tolerance mechanism to improve system availability by timely fault-detection and run-time recovery to attempt to recover a faulty chip-component or a faulty chip (Chapter 8).
- A novel protocol to provide interactive communication between the user and the system/application by bridging between the Ethernet network, which connects the system with the user interface on a Host PC, and the SpiNNaker packet-switching Communication Network, which interconnects the SpiNNaker CMPs (Chapeter 9).
- A Hardware Abstraction Layer (HAL) library of useful functions and device drivers to support application development for the SpiNNaker computing system without the need for a developer to have detailed knowledge of the underlying hardware architecture (Chapter 9).

1.5 Thesis structure

The thesis comprises 10 chapters as follows:

- Chapter 2 is a review of the literature on neural computation which forms the main motivation for the SpiNNaker project to deliver a massively-parallel multi-CMP neural network simulation engine to support large-scale simulation of biologically-realistic neural populations. This chapter explains the neural dynamics which result in information processing in a neuron, its communication to other neurons, and the resultant learned behaviour of the nervous system. It is important to understand these phenomena to understand potential applications for the SpiNNaker computing system. A reasonable explanation of neural structures and their functionality in generating an overall spatio-temporal neural information processing system would fill a complete book. However, we focus on only those concepts related to the understanding of the real-time application model for the SpiNNaker computing system.

- Chapter 3 focuses on the need to create computer simulations to explore neural information processing in neural populations (neural networks). A few software-based approaches to the simulation of large-scale neural networks are reviewed with their limitations necessitating purpose-built hardware engines for high performance. The chapter concludes with motivation for designing SpiNNaker as an Application Specific Integrated Circuit (ASIC) to support large-scale neural simulations.

- Chapter 4 highlights the objectives of the SpiNNaker computing system, its architecture, and certain important features from the programmer's point of view. The envisaged standard application model to support large-scale neural simulations with the SpiNNaker computing system is explained. Finally, a few important guidelines for software developers writing neural applications for SpiNNaker are given, along with some users' expectations acquired from our interaction with potential users.

- Chapter 5 covers our motivation for creating a system-level model for the SpiNNaker computing system. The available choices for simulating a System-on-Chip (SoC) i.e. high-level simulation vs. typical Register-Transfer-Level (RTL) modelling are described. This chapter also gives an overview of the

SystemC Transaction Level Modelling (TLM) technique as a choice for high-level simulation. Our experiences while creating a novel complete-system model for a multi-CMP system to an instruction- and cycle-accurate level are covered in this chapter. The chapter then presents some case studies performed to validate the accurate behaviour of the model at chip- and system-level, and a few experiments performed with the help of this simulation to verify the design objectives of the SpiNNaker computing system. The chapter also presents the test chip verification process with the help of the SystemC system-level and Verilog top-level behavioural models.

- Chapter 6 highlights some multi-CMP configuration challenges. Some work from the literature is presented to examine a few approaches to deal with these challenges. We describe some peculiarities of the SpiNNaker system, a truly multi-CMP system, to justify why the configuration processes used in the past are not suitable for SpiNNaker. We also identify the required features from the SpiNNaker configuration process.

- Chapter 7 explores the configuration challenges specific to the SpiNNaker architecture in the light of those already described in Chapter 6. The proposed configuration and application loading protocols for a multi-CMP system are presented in the context of the SpiNNaker system. Some experimental results are given to justify our claim that we meet our objectives.

- Chapter 8 introduces the fault-tolerance features of the SpiNNaker configuration process proposed in this thesis. Some fault-tolerance concepts are reviewed in the context of real-time computing systems and the hardware support in the SpiNNaker computing system to facilitate fault-resilience is explained. Finally, the chapter explains how the configuration process uses the SpiNNaker fault-tolerance features to make it a reliable computing system.

- Chapter 9 gives some details of the Hardware Abstraction Layer (HAL) developed to help the programmers by hiding the architectural details and to ease application development for the SpiNNaker computing system. The purpose is to make optimal use of the designed features in the SpiNNaker hardware, of which the programmer may not be aware. The chapter also describes a few features of a proposed user interface to interact with the

system.

- In the last chapter of this dissertation we conclude and summarise our research work. A few important aspects related to this research have been highlighted which could not be addressed due to paucity of time or being out of scope, and which provide a good starting point for future research.

1.6 Publications

The following publications include aspects of the work described in this dissertation:

- M.M. Khan, D.R. Lester, L.A. Plana, A. Rast, X. Jin, E. Painkras, and S.B. Furber. "SpiNNaker: Mapping Neural Networks onto a Massively-parallel Chip-multiprocessor". In Proc. Intl. Joint Conf. on Neural Networks (IJCNN2008), 2008 June 1-6 Hong Kong. (included in Chapters 4 & 5).

- M.M. Khan, J. Navaridas, X. Jin, L.A. Plana, J.V. Woods, and S.B. Furber. "Real-time Application Support for a Novel SoC Architecture". In Proc. 4th UK Embedded Forum, Southampton, UK, September 2008. (Chapters 4, 7 & 9).

- M. Khan, X. Jin, S. Furber, and L.A. Plana. "System-level Model for a GALS Massively Parallel Multiprocessor". In Proc. 19th UK Asynchronous Forum, page 19-22, September 2007. (Chapters 4 & 5).

- M.M. Khan, J. Navaridas, X. Jin, L.A. Plana, J.V. Woods, and S.B. Furber. "Configuring a GALS CMP System for Real-time Applications". In Proc. 20th UK Asynchronous Forum, September 2008. (Chapters 4 & 7).

- M.M. Khan, L.A. Plana, J.V Woods, and S.B. Furber. "System-level Model for SpiNNaker CMP System", 1st International Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'09) held in conjunction with the 4th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC2009) Paphos, Cyprus, January 25-28, 2009. (Chapters 4 & 5).

- M.M. Khan, J. Navaridas, A.D. Rast, X. Jin, L.A. Plana, M. Luján, J.V. Woods, J. Miguel-Alonso and S.B. Furber. "Event-Driven Configuration

of a Neural Network CMP System over a Homogeneous Interconnect Fabric", to appear in Proc. of Intl. Symposium on Parallel and Distributed Computing (ISPDC2009) 1-3 July 2009 Lisbon Portugal. (Chapters 4, 7 & 9).

- Luis A. Plana, Steve B. Furber, Steve Temple, Mukaram Khan, Yebin Shi, Jian Wu, and Shufan Yang. "A GALS Infrastructure for a Massively Parallel Multiprocessor". IEEE Design & Test of Computers, 24(5):454463, Sept-Oct. 2007. (Contributions in the validation process using the system-level model; Chapter 5).

- Alexander Rast, Xin Jin, Mukaram Khan, Steve Furber, "The Deferred Event Model for Hardware Oriented Spiking Neural Networks", 15th Intl. Conf. on Neural Information Processing (ICONIP2008), 2008 Nov. 25-28, Auckland, New Zealand. (Contributions in defining the SpiNNaker architecture specific to the spike communication infrastructure and deriving the results with the help of the system-level model (Chapters 4 & 5)).

- A.D. Rast, S. Yang, M. Khan, and S.B. Furber. "Virtual Synaptic Interconnect using an Asynchronous Network-on-Chip". In Proc. Intl. Joint Conf. on Neural Networks (IJCNN2008), 2008 June 1-6 Hong Kong. (Contributions in defining the SpiNNaker architecture and hardware abstraction layer functions at application level to make use of DMA operations (Chapter 4 & Chapter 7)).

- A.D. Rast, M.M. Khan, X. Jin, L.A. Plana and S.B. Furber, "A Universal Abstract-Time Platform for Real-Time Neural Networks" to appear in proc. of Intl. Joint Conf. on Neural Networks (IJCNN2009), 2009 June 14-19 Atlanta Georgia (USA). (Contributed in defining the SpiNNaker architecture specific to the axonal conductance delays in the spike communication infrastructure, setting up experiments with large population of neurons, and deriving the results with the help of a multi-CMP system-level model (Chapter 4 & 5)).

- A. Rast, S. Furber, D. Lester, S. Temple, L. Plana, E. Painkras, M. Khan, J. Wu, Y. Shi, S. Yang and X. Jin, "Abstracting both Architecture and Time: The SpiNNaker Neuromimetic Modelling Platform", ESSDERC2008, 15-19 Sep, Edinburgh.

Chapter 2

The Brain – Nature's Masterpiece

"The human brain is by far the most complex structure in the known universe. The extraordinary properties of this three or so pounds of soft tissue have made it possible for Homosapiens to dominate the earth, change the course of evolution through genetic engineering, walk on the moon, and create art and music of surpassing beauty. We do not yet know the limits of the human mind and what it can accomplish" Richard F. Thompson [Tho00]

2.1 Introduction

SpiNNaker's target application is the simulation of biologically-realistic neural networks. Neural networks are characterised by parallel distributed processing employing a massive number of small independently functional processing units (neurons) with a tremendous amount of connectivity. Several key properties of "real" neural networks drive the design of SpiNNaker's application-specific architecture. It is, therefore, important to understand a neuron's functional behaviour in order to comprehend the intended applications for the SpiNNaker massively-parallel computing system. Neurons communicate through spikes: short-duration impulses [DA01] (Figure 2.1[1]). It is usual to abstract the spike to an instantaneous pulse, or event, triggered when the neuron reaches a certain threshold value [Me98]. The neuron's spike is characterised by a temporal delay on a millisecond scale during its development and firing due to its biochemical properties.

[1]The diagrams from publications are included with the authors' permission.

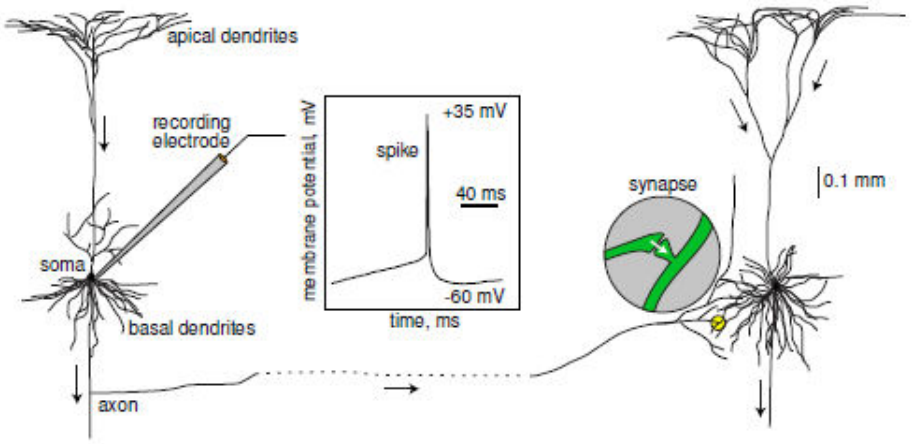

Figure 2.1: Neural Structure (from [Izh07], fig 1.1 page 2).

The spiking behaviour of neurons with the associated temporal characteristics is called the neural dynamics of the nervous system and controls its functional behaviour including stimulus-response and learning.

This chapter will first explore briefly some neural structures to make the later discussion accessible to readers who may have a limited knowledge of the nervous system. This is important to introduce these concepts from a neural computation point of view as these will be referred a number of times in the later chapters. The chapter then focuses on a few neural dynamics phenomena which contribute together to the information processing mechanism in neurons and then to a collective intelligent behaviour from the nervous system. We culminate this discussion by describing a valuable effort to capture neural dynamics using a mathematical model to help reproduce this behaviour within the computer simulations. The chapter will end with an introduction to learning dynamics in neural networks. We shall not go into the details of these concepts to avoid unnecessary digressions as this research is not directly based on neural dynamics.

2.2 Basic Neural Structure

The neurons (Figure 2.2) are the basic functional unit of the nervous system [Tho00]. Neurons are like other body cells in most respects. However, they are specialized in their behaviour in order to process and transmit information to other neurons

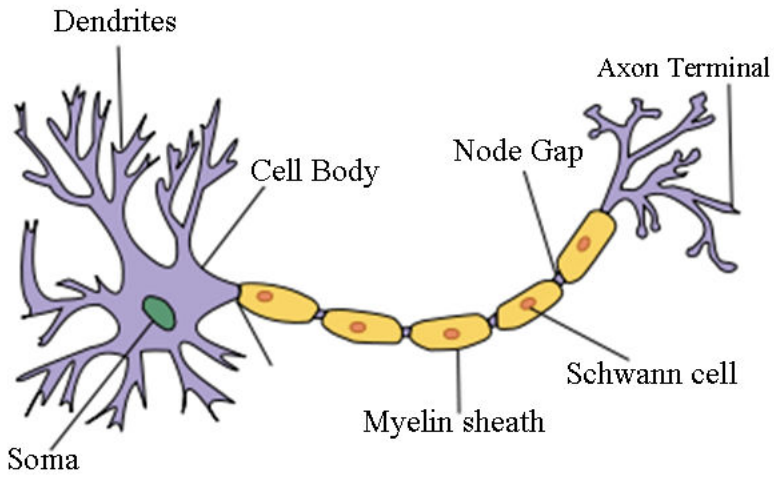

Figure 2.2: A Typical Neuron (from [Wik09a]).

or muscles. The brain's behaviour, as a whole, depends on the way its neurons are organized, so to understand the brain it is important to understand neurons and their information processing mechanisms. A neuron's functionality is defined by its microstructure and connectivity to other neurons. One major characteristic of neurons that differentiates them from the other body cells is the structure of their cell membrane, which is made specifically to transmit and receive information. Neurons are specialized in information processing, utilizing special electrophysical and chemical processes [Tra02]. There are many different types of neuron based on their size, shape, and physiological properties. However, many features of neurons can be generalized to almost all types of neuron.

The following sections describe some important parts of a neuron which contribute to its functional behaviour.

2.2.1 Membrane

The neural membrane is made of organic fluid (phosphoric acid and fatty acid), about 3-4 nm in thickness [DA01], that keeps a neuron intact in the ionic water solution inside and around it. The ionic solution contains ions which are mostly sodium (Na^+), potassium (K^+), calcium (Ca^{+2}), and chloride (Cl^-). The membrane is impermeable to most charged molecules, which causes it to behave like a capacitor separating charges along its interior and exterior surfaces. The cell membrane contains some protein molecules scattered throughout it. These

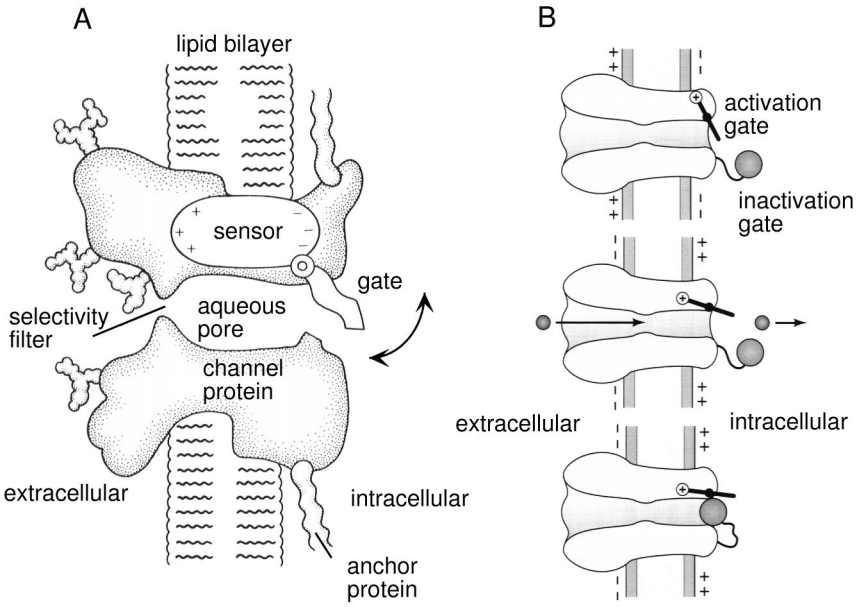

Figure 2.3: Ionic Channels in Neural Membrane (from [DA01], fig. 5.8 page 169)).

molecules are the chemical receptors of charged ions and are called *ion channels* as shown in Figure 2.3. Charged ions attach to the appropriate protein molecules on the cell membrane and may cause various changes in both the membrane and the inner process of the neuron [Tho00]. There can be more than a dozen types of ion channel in a cell membrane, with a total number of channels in a neuron's membrane ranging from hundreds to thousands [Tho00]. Most of these channels form valves to allow specific ions to enter or leave the cell body and are named after the ions they allow to pass, i.e. a Na^+ ion channel will let only Na^+ ions to pass through it. Some channels are always open and are called *leakage channels* while the other channels are normally closed and are activated by certain changes in the membrane.

2.2.2 Axon

Each neuron has only one axon as an output terminal to other neurons, gland cells or muscles. The axon may branch and send multiple fibres to attach to the other neurons through its *axon terminals* (Figure 2.2). The axons are filled with tiny tubes running the length of the axons from the cell body to the synaptic terminals called *microtubules*. Chemical substances called *neurotransmitters*, which are

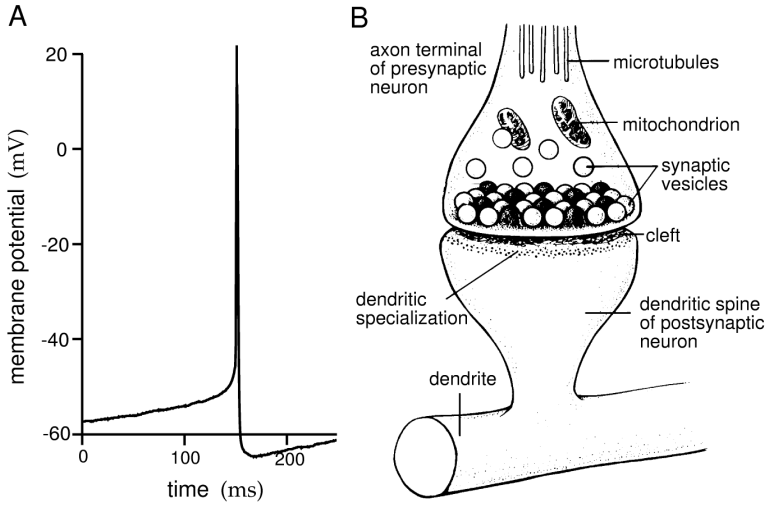

Figure 2.4: Inter-neuron Synapse (from [DA01], fig 1.2 page 6).

produced in the cell body, are transported to the axon terminals with the help of these microtubules. The chemicals move in both directions with the help of a fast and a slow process with a transportation speed of 10-20 millimetres per day and 1 millimetre per day respectively. The larger axons are covered with a sheath of fatty insulation called *myelin* (Figure 2.2). Myelin considerably increases the speed of conduction in an axon [Tho00]. Once an axon is fully developed, it is covered by many layers of myelin.

2.2.3 Dendrite

Dendrites are thought to be an extension of the cell body's receptive surface. These are the fibres around neuron (Figure 2.2) which give a neuron its typical tree-like shape. They may range from a few short fibres to a huge mass of entangled bushes. A typical neuron can have 10,000-100,000 dendrites. The axons from one neuron can attach to the cell body of another neuron directly or through these dendrites to form *synapses* (Figure 2.1). The dendrites of many neurons are covered with thousands of little extensions called *dendritic spines*. The spine is the *postsynaptic* part of the synapse made with the *presynaptic* axon-terminal from some other neuron [Tho00].

2.2.4 Synapse

One way that neurons are different from other body cells is that they make synapses with other neural tissues with the help of their axons. Synapses (Figure 2.4) are the point of functional contact formed by axons to the other neurons at their dendrites or cell bodies. Synapses can be divided into two basic types: electrical and chemical synapses [Tho00]. The mammalian brain predominantly contains chemical synapses. The neuron contributing its axon to a synapse is termed as a *presynaptic* neuron while the neuron on whose dendrite or cell body the axon is attached is called the *postsynaptic* neuron. Axon terminals contain a large number of chemical pockets called *vesicles* which are filled with *neurotransmitters*. The attachment point on the postsynaptic neuron forms a dense staining band that defines the extent of the the synapse. In between the pre- and postsynaptic contact points, there is a space of about 20 nm called *synaptic cleft*. When a synapse is active, the vesicles open and release neurotransmitters into the synaptic cleft. These are received by chemical receptor molecules on the surface of postsynaptic neurons. A synapse can either be *excitatory* or *inhibitory* depending on the type of neurotransmitter being released by the presynaptic neuron. The excitatory synapses increase the activation of target neurons while the inhibitory synapse reduce their activation.

Various types of neurotransmitter with specific properties are found in the nervous system. These neurotransmitters regulate neurotransmitter-gated ion channels which, in turn, regulate the ionic conductances of the membrane through the binding of particular neurotransmitters. On receipt of a specific neurotransmitter, these channels allow ions of specific size and shape to pass through them. Each neurotransmitter affects the receiving neuron in a different way.

2.3 Membrane Potential

The concentration of ions is not the same within and outside a neuron as shown in Figure 2.5. This difference between the ion concentration causes a concentration gradient across the neuron's membrane. The concentration of K^+ is much higher inside the neuron than in the the fluid outside the cell. The leakage-channels allow the K^+ from the ionic solution inside the neuron to move outside to balance its concentration under a *diffusion force* caused by the concentration gradient. The solution inside the neuron contains negatively charged protein ions P_{2-} which

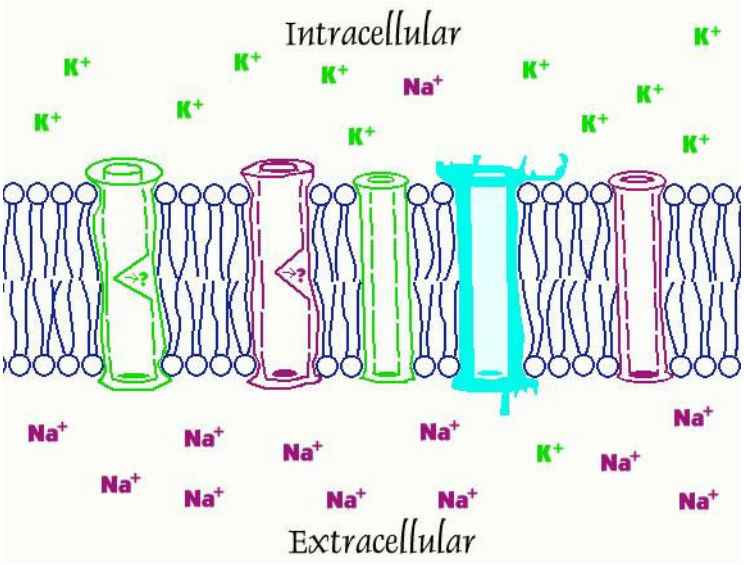

Figure 2.5: Ionic Channel Interplay to Develop Membrane Potential (from [Wik09d]).

always remain inside the cell because of their large size [Tho00]. The diffusion process leaves an excess of negative charge inside the neuron and an excess of positive charge outside the cell body which causes a strong *electric potential* across the cell membrane. The electric force is in the opposite direction to the diffusion force and tries to stop K^+ flow. The two forces, i.e. the ion concentration gradient and the electric potential, will eventually balance each other to stop further transportation of K^+ ions from inside to outside the cell membrane. The neuron settles at an equilibrium state with a concentration of K^+ ions of 400 and 20 millimoles per litre inside and outside the cell respectively. The resting membrane potential due to K^+ can be found using the Nernst equation [Tho00] as:

$$V = \frac{RT}{kF} log \frac{[I^+]_{in}}{[I^+]_{out}} \tag{2.1}$$

where k, R and F are constants, T is the absolute temperature and $[I^+]$ is the concentration of positive ions. At a normal temperature of 18^0 Celsius $\frac{RT}{kF} = 58$ and the Nernst equation gives a resting potential of K^+ [Tho00] as:

$$V = 58 log[\frac{20}{400}] = -75.46mV \approx -75mV \tag{2.2}$$

Similarly, the other ion concentrations, such as those of Na^+, Cl^- and Ca^{+2}, are greater on the outside of the cell membrane, however, these do not contribute substantially to the resting potential as the Na^+, Cl^- and Ca^{+2} ion channels are not affected too much by the concentration diffusion force. These are mostly voltage-gated or neurotransmitter-gated channels as compared to the K^+ channel which are mostly diffusion-gated leakage channels. The Nernst equation can be extended to take into account the effect of these channels as well [Izh07], eventually leading to the resting potential of a neuron, which is typically about $V_{rest} = -65mV$ [Tra02].

2.4 Action Potential

As discussed earlier, the membrane at rest is slightly permeable to Na^+ ($\frac{1}{20}th$ the permeability of K^+) due to leakage channels. When a large quantity of neurotransmitter is released by the presynaptic neuron or a small quantity of it released by a few presynaptic neurons in a short span of time, this can trigger a stronger response in the postsynaptic neuron's membrane potential. If the neurotransmitter is excitatory, it will move the potential difference in a positive direction (from -65mV to -30mV - *threashold potential*) and activate the cell membrane, developing an action potential across the membrane. This opens all $Na+$ channels suddenly, making the membrane permeable to Na^+. The diffusion force due to the strong concentration of Na^+ outside the cell body pushes Na^+ inside the cell (Figure 2.5). As there are more negatively charged ions (protein ions) inside the cell, the positive Na^+ ions rush inside due to electrical force as well. Both forces, i.e. the diffusion force and the electrical force, act in the same direction to bring the Na^+ inside the cell body with 500 times more permeability for Na^+ than the one it has in the resting state. In a short period, the membrane potential reaches a Na^+ equilibrium value of +50mV (an increase of $\approx 100mV$ in the membrane potential from its resting potential), which closes the Na^+ channels. At the same time a few voltage-gated K^+ channels which are normally closed at the resting potential are opened, letting K^+ move outside more freely than at rest. This removes K^+ from the cell bringing the membrane potential towards its resting potential. As the opening and closing of the K^+ channels is slower than the Na^+ gates, the K^+ gates remain open while the $Na+$ gates are closed, as a result of which the membrane potential goes toward -75mV (more

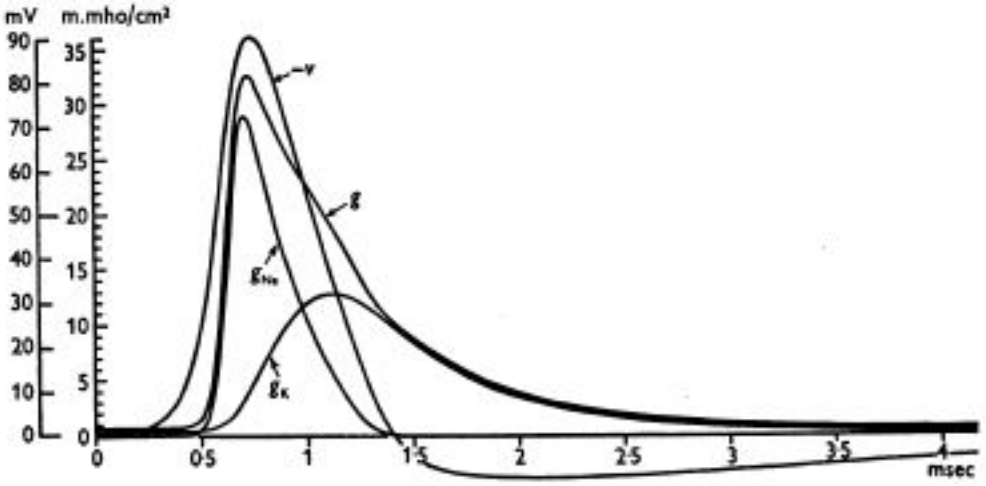

Figure 2.6: Membrane equivalent conductance (g) and voltage (v) as a result of interplay between Na^+ conductance (g_{Na}) and K^+ conductance (g_K) as measured by Hodgkin and Huxley [HH52] (they used a resting potential of 0 mV instead of -65 mV for computational simplicity).

negative than the resting potential), i.e. towards the K^+ equilibrium potential. This state of more negative voltage than the resting potential remains for a few milliseconds and is called the *afterpotential* state or *hyperpolarization*.

The few voltage-gated K^+ channels close eventually and the potential returns back to the normal resting (-65mV) voltage. The period when the Na^+ channels are open, and the spike of action potential develops and decays, is called the *absolute refractory period*. During this period the neuron can not be electrically stimulated to generate another action potential [Tho00]. After the spike, i.e. in the afterpotential period, the neuron can be electrically activated, however, it requires a very strong electrical stimulus. Normally neurons do not spike during this period. This is called the *relative refractory* period. The hyperpolarization of the neuron relative to its resting voltage causes the voltage-dependent K^+ channels to close and the voltage-dependent sodium channels to deactivate, eventually reinstating the normal resting potential of the neuron. Figure 2.6 shows the interplay of the ionic conductances due to the opening and closing of the ionic channels during the process of action potential. The graph labelled V shows the resultant membrane potential as an accumulated voltage across the membrane.

With the repeated generation of action potentials, resulting in repeated outflow of $K+$ and inward flow of Na^+, the $K+$ concentration will decrease and

$Na+$ concentration will increase within the cell. This will ultimately reduce the probability of generating an action potential. At this stage, the action potential can be generated with the help of yet another type of ion channel called an *ion pump*, which can transfer ions against their concentration gradient. The process is expensive as a neuron requires energy for this process [Tra02].

2.5 Synaptic Transmission

Information processing in the brain is dependent on the transmission of signals between the neurons. Chemical synapses are the main source of information transfer in the nervous system [Tra02]. The cell membrane has a relatively high resistance and accumulates charge on both sides creating a capacitor effect. The action potential is generated at the stem of the axon where it starts from the neuron's body. As the Na^+ moves into the axon at the site of the action potential, the region closer to the site begins to become relatively less negatively charged from its resting potential due to N^+ ions, a process called *depolarization*. The rate of depolarization depends on the membrane capacitance. The patch of membrane immediately next to the place where the Na^+ gates open will then reach the $Na+$ gate opening threshold voltage. In this way the action potential continuously moves down the axon until it reaches the axon terminal which contains normally closed Ca^+ channels. The arrival of the action potential opens these channels briefly, causing an influx of Ca^+ ions due to the diffusion gradient as Ca^+ ions are more in number outside the cell membrane than inside it. The Ca^+ ions then trigger the release of neurotransmitters. The released neurotransmitter at the synapse enters the cleft and attaches the postsynaptic membrane.

As a result of excitatory neurotransmitters the neurotransmitter-gated Na^+ channels open in the postsynaptic neuron. The postsynaptic neuron becomes depolarized and if reaches a threshold potential, the voltage-gated Na^+ channels will open, generating an action potential and so on. This process is time consuming as it takes a few milliseconds from the time the action potential reaches the presynaptic axon terminal to the point where the postsynaptic neuron is depolarised. It is believed that the sodium channels are situated closer to the axon stem and along the axon, as the rest of the cell body and the dendrites are covered with axons from other neurons and thus act as if myelinated. Because of this reason, the action potential is generated closer to the point the axon emerges

from the cell body and travels along the axon towards its terminal rather than moving in the opposite direction [Tra02]. Normally, the activation of a single excitatory synapse on a neuron will not cause it to develop an action potential. It may require a number of synapses to be activated together in a short span of time and influence the postsynaptic neuron together - a phenomenon called *spatial summation*. If the synapses are far apart, they may not generate an action potential on being activated together. However, if they are activated repeatedly at a fast enough rate, they will sum over time and generate an excitatory postsynaptic potential (EPSP) to cause an action potential in the postsynaptic neuron. This phenomenon is called *temporal summation* [Tra02].

2.6 Neural Networks

Each neuron connects with on the order of 1000 other neurons (some may receive even more than 100,000 input connections). A small number of interconnected neurons can exhibit complex behaviour and information processing capabilities not present in a single neuron [DA01, Tra02]. There is still little understanding of such non-linear interacting systems which are characterised by additional information processing capabilities beyond that of single neurons, such as presenting information in a distributed manner. A combination of such networks in a specific area of the nervous system is able to perform even more complex information processing tasks. It is the neural interaction in large population of neurons that enables their processing abilities different from a single neuron. This forms a system of interacting processing units with emergent properties. Emergence is probably the most defining property of neural networks which distinguishes these from parallel computing in classical computer science [Tra02]. Interacting real-time systems have unique properties beyond combined properties of single processors and the outcome may not be deterministic, while in a typical parallel computing system we distribute a large and complex job across independent algorithmic threads to speed up the processing but to get the same result. Emergent systems, however, are rule-based systems. Neural networks are also governed by a finite set of rules or fundamental laws [Tra02]. These rules are not fully understood in case of the nervous system and some people are convinced that a simple set of rules explaining brain functions might never be discovered, or even after discovering some of these rules we might not have sufficient understanding of

this most complex emergent system in the universe. However, there is enormous progress to this end, and today we know much more than we knew a decade ago.

2.7 Neural Computation

Brain imaging techniques, such as fMRI (Funcational Magnetic Resonance Imaging) etc, have enabled us to discover activities in various regions of the brain while performing a specific mental task. We can spot the temporal interaction among neurons or groups of neurons as a result of a particular activity and trace the flow of information from stimulus to a resulting response. Such techniques have provided a huge amount of useful data about the information processing mechanisms inside the brain. New quantitative hypotheses about neural functions are being formulated inviting more specific experimental analysis to test these hypotheses. For a realistic understanding of the brain's functionality, it is important to devise hypotheses precisely and to test them experimentally to verify or disprove them [Tra02]. We need to discover the brain's behaviour at two levels i.e. at the individual neuron level to discover the information processing mechanism and the way it interacts with other neurons, and at the neural network level to understand the minimal features contributing to certain emergent properties of these networks [Tra02].

One such empirical hypothesis and a resultant model was presented by Alan Hodgkin and Andrew Huxley in 1948 [HH52]. They quantitatively described the form of action potentials using numerical equations long before ion channels were known or the details of the neural spiking process were measured directly [Izh07]. The Hodgkin-Huxley mathematical model is the most accepted one to capture neural dynamics in biological neurons [Izh07]. They formulated this model based on their experiments on the giant axon of a squid. As per this model, the number of open ion channels is proportional to an electric conductance g_{ion} and the movement of ions through these channels (electric current) I_{ion}. The resultant equilibrium potential for a particular channel E_{ion} created across the membrane is relative to its resting potential V. The relationship between the electric potential, the current and the conductance is given by Ohm's law [Tra02] as

$$I_{ion} = g_{ion}(V - E_{ion}) \tag{2.3}$$

As described before, three types of channel contribute to the changes in the membrane potential, i.e. the leakage channels and the voltage dependent K^+ and Na^+ channels. The voltage controlled channels have varying conductances which are dependent on the number of channels opened at a particular time. Hodgkin and Huxley attached an empirical behaviour to these channels which they found from their experiments. They defined the conductances of these channels using three variables chosen appropriately to approximate their behaviour to the experimental data. The variables n, m and h describe the activation of K^+ and $Na+$, and the inactivation of Na^+ channels respectively. The effect of these variables on the channel conductances is given by,

$$g_K = g_K n^4 \tag{2.4}$$

$$g_N a = g_{Na} m^3 h \tag{2.5}$$

These variables represent the number of activation or inactivation gates such as the voltage-gated K^+ current with four activation gates (represented as n^4), and the voltage-gated Na^+ current with three activation gates (m^3) and one inactivation gate (h^1) [Izh07].

The dynamics of the neuron with respect to each channel is captured by,

$$dx/dt = -1/(\tau x(V)) * [x - x_0(V)] \tag{2.6}$$

where x can be substituted by each variable i.e. n, m or h in turn. Hodgkin and Huxley selected these variables to get a reasonable fit to the experimental data. This equation describes the voltage dependence of K^+ and Na^+ channels. The leakage channel is static with constant conductance g_L.

Figure 2.7 shows an equivalent electric circuit with three conductances to represent three channels i.e. the leakage channel, the K^+ channel and the Na^+ channel with their own batteries (the ion potential difference due to the concentration gradient which tends to move the charges in the direction of lower concentration). The electric charge stored by the neuron is formally represented by the voltage and its capacitance C in parallel with the three resistors as shown in Figure 2.7. The combined effect of all these components can be formalized by Kirchhoff's law as,

$$C\frac{dV}{dt} = -\Sigma I^{ion} + I(t) \tag{2.7}$$

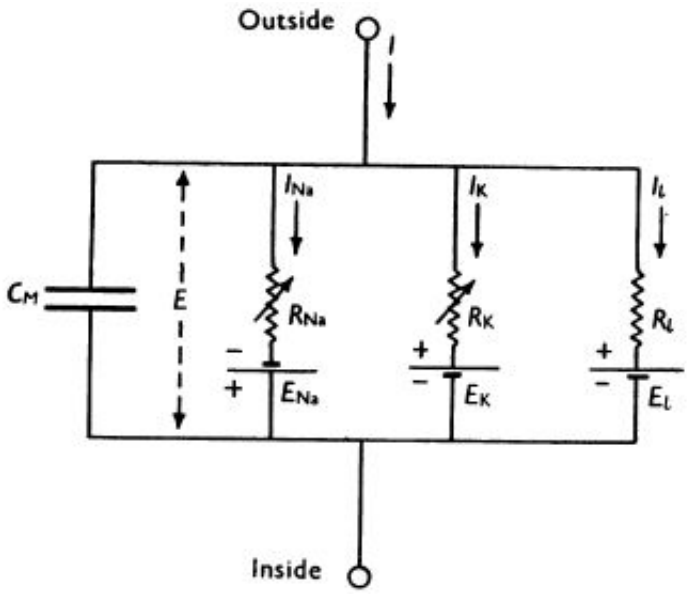

Figure 2.7: Equivalent Electric Circuit for Ionic Channels used by Hodgkin and Huxley (from Hodgkin and Huxley [HH52]).

This first order equation describes the change in membrane potential with time. $I(t)$ is an external current such as the current from neurotransmitter-gated ion channels. By combining the equations together [Izh07] we get,

$$\frac{dV}{dt} = -g_K n^4 (V - E_K) - g_{Na} m^3 h (V - E_{Na}) - g_L (V - E_L) + I(t) \tag{2.8}$$

The variables n, m and h can be computed by:

$$\frac{dn}{dt} = \alpha(V)(1 - n) - \beta(V)n \tag{2.9}$$

$$\frac{dm}{dt} = \alpha(V)(1 - m) - \beta(V)m \tag{2.10}$$

$$\frac{dh}{dt} = \alpha(V)(1 - h) - \beta(V)h \tag{2.11}$$

where

$$\alpha_n(V) = 0.01 \frac{10 - v}{exp(\frac{10-V}{10}) - 1} \tag{2.12}$$

$$\beta_n(V) = 0.125 exp(\frac{-V}{80}) \tag{2.13}$$

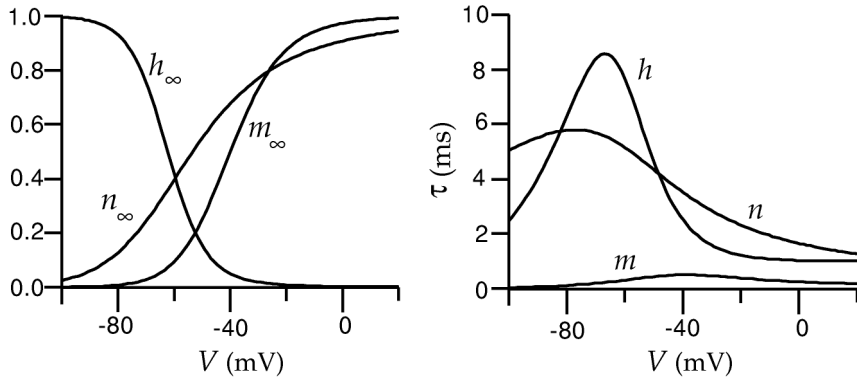

Figure 2.8: Empirical Variables used by Hodgkin and Huxley (from Hodgkin and Huxley [HH52]).

$$\alpha_m(V) = 0.1\frac{25 - V}{exp(\frac{25-V}{10}) - 1} \tag{2.14}$$

$$\beta_m(V) = 4exp(\frac{-V}{18}) \tag{2.15}$$

$$\alpha_h(V) = 0.07exp(\frac{-V}{20}) \tag{2.16}$$

$$\beta_h(V) = \frac{1}{exp(\frac{30-V}{10}) + 1} \tag{2.17}$$

The parameter values were set by Hodgkin and Huxley by shifting the membrane potential approximately +65mV (representing the -65mV resting potential at 0mV as a reference voltage) to bring $V_{rest} \approx 0V$ for the sake of convenience [Izh07]. By restoring the membrane potential back to its original value, we get $V_{rest} \approx -65mV$. The shifted values of equilibrium ion potentials are, $E_K = -12mV, E_{Na} = 120mV, E_L = 10.6mV$. While the maximal conductances values are, $g_K = 36mS/cm^2$, $g_{Na} = 120mS/cm^2$, $g_L = 0.3mS/cm^2$.

The capacitance around the membrane $C = 1\mu F/cm^2$ and the applied external current is $I = 0\mu A/cm^2$. Functions $\alpha(V)$ and $\beta(V)$ define the transition rates between the open and closed states of the channels. In Equation 2.8 the ionic currents are K^+ (I_K), Na^+ (I_{Na}) and leakage (I_L). The leakage current is the Ohmic leakage current carried mostly by Cl^- ions [Izh07]. In a standard form the ionic variables (n, m and h) are are shown in Figure 2.8.

In order for a neuron to generate an action potential, the densities of these channels have to exceed a certain threshold depending on the temperature. The

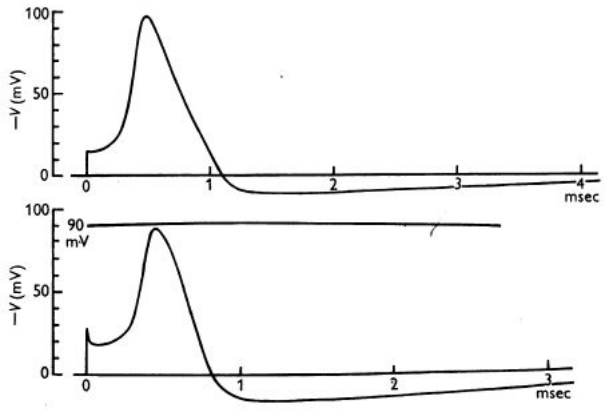

Figure 2.9: Threshold potential as noted by Hodgkin and Huxley [HH52] at 18^0 C (bottom graph) and 20^0 C (upper graph) (they used a resting potential of 0mV instead of -65mV for computational simplicity).

behaviour of this model as measured by Hodgkin and Huxley is shown in Figure 2.9.

2.8 Temporal Dynamics

Neurons communicate with the help of action potentials, spikes, which travels as an all-or-nothing boolean operation i.e. the information is passed by the presence or absence of a *pulse*, but not by its size or shape [Me98]. This theory generates a very important question: How does the brain process information and store/retrieve it if the only means of communication is the action potential pulse? Most researchers now believe the answer to this question lies in the *spike timing*, which may be used by our nervous system to *code/decode* information [Me98, Izh07, DA01]. Besides this, the neural dynamics in the nervous system is tied to a temporal behaviour which is dictated by the biochemical processes inside neurons. This sets an upper bound on the spike frequency from a neuron in the domain of a few 100Hz (i.e. a few spikes every second or normally a spike after a few milliseconds) as shown in Figure 2.10. An action potential lasts for about 1 ms. After the action potential, the process of absolute refractory and relative refractory prohibits a neuron from firing for a few milliseconds [DA01]. This implies that a neuron can spike at a frequency not more

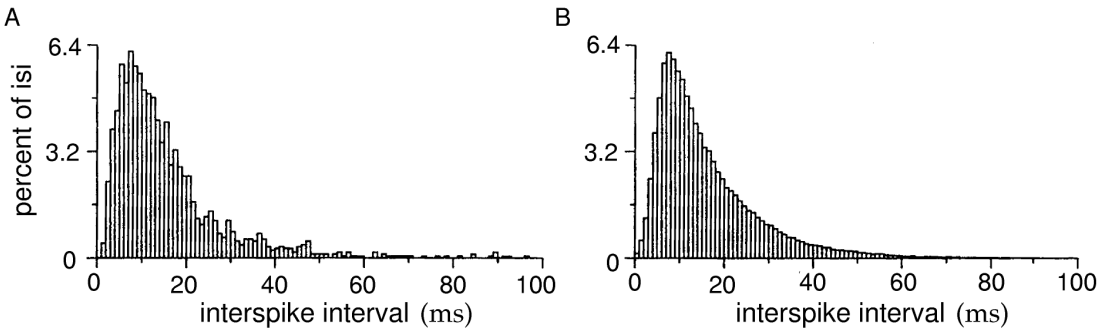

Figure 2.10: Inter-spike Interval (isi) (from [DA01], fig 1.15 page 33).

than a few 100s of Hz. The process of spike transmission in the axon also involves a delay called the *axonal conduction delay* due to the physiological process of opening the ionic channels along the axon's length (ranging from a few mm to a metre [Tho00]). The spike transmission velocity in a myelinated axon has been measured as $\approx 1m/s$, while in the non-myelinated axon it is only $\approx 0.15m/s$, which introduces an average axonal delay of up to 10ms [EMIE04] (and it may be more in some cases). The process of post synaptic potentiation as a result of spike receipt also involves some delay in opening ion channels. It has been discovered that a neuron performs computation on localized regions of its dendritic tree (dendritic compartments [Me98, Izh07, DA01]), which involves a further delay in passing on the pulse to the cell body and thus generating an action potential. Figure 2.10 from Dayan et al. shows the inter-spike interval of a typical pyramidal neuron in mammalian neocortex and the spike rate of cortical neurons in normal circumstances. We include this figure to give an idea of the spike frequencies in the nervous systems (the proportion of active neurons in the nervous systems at any one time are only ∼0.01-1% [DA01]).

2.9 Adaptability and Learning

Activity-dependent synaptic adaptability or *synaptic plasticity* is widely considered to be the basic phenomenon of learning and believed to be the primary contributor to the development of neural circuits [DA01]. Donald Hebb in 1949 proposed a learning mechanism which has since become known as the *Hebbian Learning* rule. As per this rule, if neuron A often contributes to the firing of neuron B, then the synapse from neuron A to neuron B should strengthen [DA01].

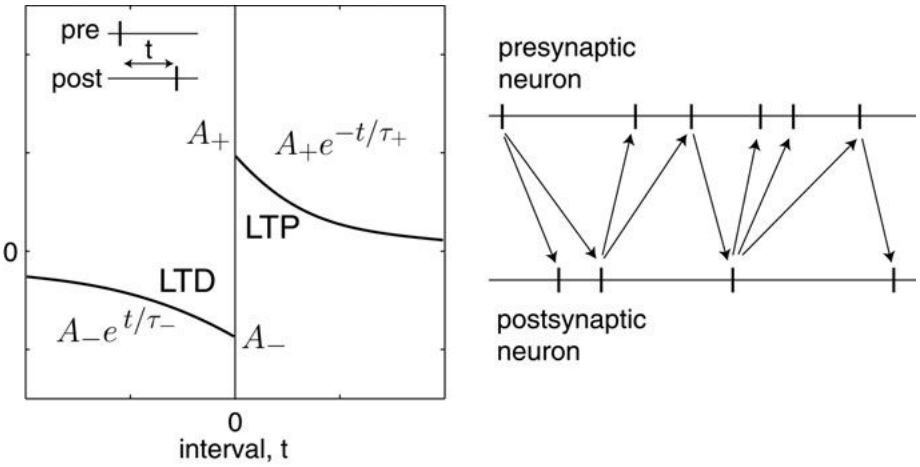

Figure 2.11: Spike Timing Dependent Plasticity (STDP) (from [EMIE04], fig. 3).

In other words, the synaptic changes are in proportion to the correlation or covariance of the activities of the pre- and postsynaptic neurons. Experimental results in many parts of the brain have proved to be activity-dependent processes. Experimental results show that generally in our nervous systems, high-frequency stimulation for a long-enough period causes *synaptic potentiation* (an increase in synaptic weights), and long-lasting low-frequency stimulation induces *synaptic depression* (a decrease in the synaptic weights) [DA01]. These effects are termed *long-term potentiation* (LTP) and *long-term depression* (LTD). The synaptic strength is affected by these stimulations with a transient or long-lasting effects. As a general guideline, the changes that persist for up to tens of minutes or longer are LTP and LTD while very long-lasting effects are caused by protein synthesis [DA01]. It is believed that the postsynaptic concentration of Ca^+ plays a major role in both LTP and LTD. The process as a whole is termed *synaptic plasticity* and contributes to the *learning* in the nervous systems. Basic Hebbian learning rule can be described by the following relation:

$$\tau_\omega \frac{d\mathbf{w}}{dt} = v\mathbf{u} \tag{2.18}$$

where τ_ω is the time constant controlling the rate of weight change, $\mathbf{w}$ is a vector defining all the synaptic weights in the network, while v is the postsynaptic activity evoked directly by the presynaptic activity $\mathbf{u}$.

There are many forms of Hebbian learning process found in the literature,

however, we shall discuss only timing dependent plasticity, called *Spike Timing Dependent Plasticity* (STDP), as an example of a learning process. Experiments have shown a strong relationship between the timing of pre- and postsynaptic action potentials [DA01], and this is attracting the attention of researchers. As per this rule, synaptic plasticity occurs only if the difference in the pre- and postsynaptic spike times falls within a window of roughly $\pm 50ms$ [DA01]. The sign (potentiation or depression) depends on the order of stimulation i.e. a presynaptic spike that precedes a postsynaptic action potential induces LTP, respectively a presynaptic spike that follows a postsynaptic action potential will produce LTD. The rule is directly in line with the basic Hebbian hypothesis as a synapse is strengthened only when a presynaptic action potential precedes a postsynaptic action potential. Figure 2.11 shows the relative timing of the pre- and postsynaptic action potentials and its effect on the amount and type of plastic modification. The figure shows *in vivo* experimental results from a pair of cortical neurons of a tadpole.

The following relation from Dayan et al. [DA01] describes the STDP process in a neural network:

$$\tau_\omega \frac{d\mathbf{w}}{dt} = \int_0^\infty d\tau (H(\tau)v(t)\mathbf{u}(t-\tau) + H(-\tau)v(t-\tau)\mathbf{u}(t)) \tag{2.19}$$

where τ is the temporal difference between the times when the firing rates of pre- and postsynaptic neurons are evaluated, function $H(\tau)$ determines the rate of synaptic change due to postsynaptic activity separated from presynaptic activity by time τ. The relation is based on the assumption that the rate of synaptic change is proportional to the product of the pre- and postsynaptic rates which is in accordance with the basic Hebbian rule. The first term on the right side of the relation represents LTP and the second represents LTD for a positive $H(\tau)$ with positive τ and negative with a negative τ. Again u and v are the pre- and postsynaptic activities. while the $H(\tau)$ is given by [EMIE04] as:

$$H(\tau) = A_+ e^{-t/\tau_+} \quad for \quad t > 0, \tag{2.20}$$

$$H(\tau_-) = A_- e^{t/\tau_-} \quad for \quad t < 0 \tag{2.21}$$

where $A_\pm$ is the weight-dependent coefficient. A constant value of these coefficients introduces additive learning.

2.10 Summary

In order to understand a neural simulation application, it is important to understand the dynamics of biological neurons and their collective behaviour as a population of neurons interconnected to form a neural network. Our brain is made of billions of neurons - functionally independent processing units with a tremendous amount of connectivity. The neuron's behaviour is dictated by its electro-physiological properties controlled by chemical ions inside and around its cell body. The concentration of these ions is maintained by the concentration diffusion and electric forces at the neuron's normal resting potential. Stimuli to a neuron in the shape of neurotransmitters cause an action potential - a spike or pulse. Neurons communicate with each other and with the muscles/glands using these spikes. All our body movements, responses to our senses and learning/memories are controlled with these spikes. Much is known about the neural and learning dynamics in the nervous systems. However, a lot remains to be discovered, especially the emergent behaviour of neural networks and phenomena of short/long term information storage. Many mathematical models have been proposed based on empirical hypotheses to capture the neural dynamics in the nervous systems. There is a need to explore more using these models and to refine them based on the *in vivo* experiments to understand the collective behaviour of large neural networks.

Chapter 3

The Brainbox – Learning From Nature

"The computer models (neural simulations) allowed us to perform experiments that are impossible (physically or ethically) to carry out with animals"

Eugene M. Izhikevich and Gerald M. Edelman [IE08]

3.1 Computer Simulations

Easy access to increasingly powerful computing systems, together with advances in the computational processes needed to capture accurately the behaviour of very complex systems, has made computer based modelling an attractive research tool [HG93]. Investigating complex phenomena with the help of computer simulations has become an established practice in science and engineering [HE88]. In physics, chemistry and biology, computer-based simulations are used to research problems in complex processes related to dynamical systems, large-scale structures, protein mechanics and cell metabolism etc. From heavy mechanical industry to medical sciences, computer simulations are used to refine the costly processes that cannot be carried out repeatedly with actual subjects. In aircraft design, building and flying a prototype to evaluate design changes is hazardous and expensive, besides it being extremely difficult to record accurately the desired effects. In general, we are not in the fortunate position of being able to test under all intended environmental conditions as these are under the control of nature.

Similarly, in medical sciences many animals 'sacrifice' their lives in the pursuit of enhancing human life expectancy. Computer simulations are the most suitable alternatives for such experiments and can, in some cases, reduce cost as well as avoid those of animal subjects [HE88]. One advantage of computer modelling is that a specific design feature can be tested against a particular environmental effect in isolation (or a combination of only desired effects), which is not possible in reality. For example, the effect of side wind at varying speed on the design of an aircraft can easily be simulated, which may be impossible to achieve in flight tests.

Human imagination and thinking has been a mystery for many centuries. The quest goes back to ancient times when we assumed the existence of some spiritual power (soul) to control our imagination [RCAT97]. To the Egyptians, the thinking process was controlled by the heart. So the brain was removed from a mummy's skull believing it to be of no significant use in the second life. During the last century, increasing discoveries about the mammalian nervous system at the behavioural level invited research communities from multiple disciplines to explore the brain. The successful use of neural networks (NN) by Artificial Intelligence (AI) and psychology, such as in the areas of character and speech recognition, adaptive systems, and robotics etc., brought a lot of public and commercial interest into the research. The 21^{st} century was expected to be the era of intelligent machines with human-like behaviour. However, we could not sustain the expected pace in AI due to our lack of understanding of "intelligence" before attempting to build an "intelligent machine" [HSB04]. Further discoveries in neuroscience forced researchers in computer science to pay more attention to understanding the brain [Izh03b] before creating "brain-like" artefacts.

With multi-disciplinary collaboration, computer-based modelling is starting to converge with the behaviour of *in vivo* neurons. Neural simulations have started to explore biologically-plausible models of the neural network's behaviour. Figure 3.1 illustrates a multi-disciplinary interaction among researchers for the use of neural simulation. Acquiring an insight into the brain's functionality, making an accurate mathematical model based on those discoveries, and then using these models to develop computer simulations of *in vitro* experiments is becoming the most viable option for exploring the mysteries of the brain [Tra02]. A two-way process wherein computer scientists and neuroscientists educate each other is facilitating research in the two disciplines. As a result the end users in

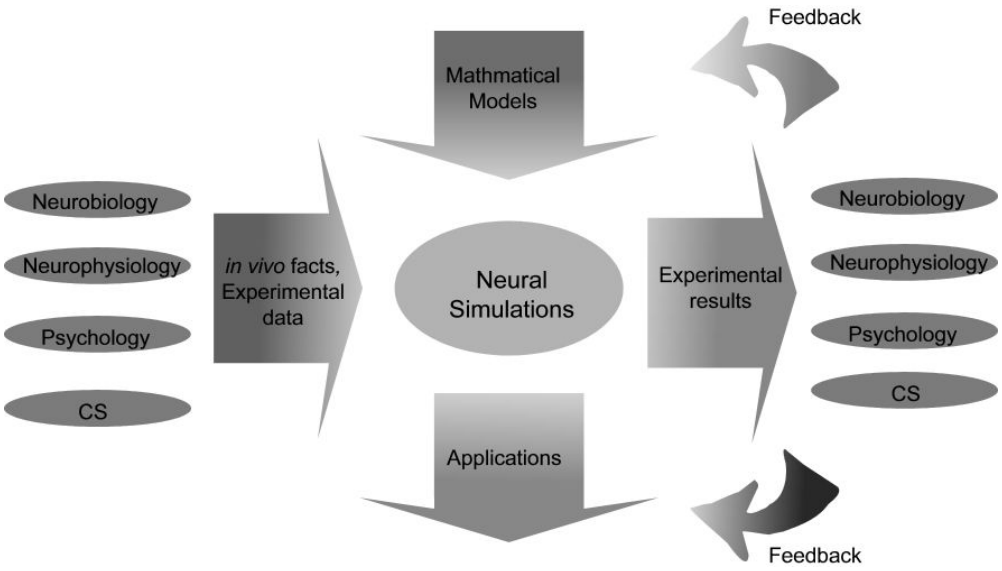

Figure 3.1: Multi-disciplinary Interest in Neural Simulation (adapted from [Tra02], fig 1.2 page 6.)

the fields of medical science, computer applications, robotics, AI, and psychology etc. are benefiting from these advances.

This chapter focuses on the need for creating computer simulations of populations of neurons (neural networks). Software-based neural simulation approach is explored in a review of the literature understanding its limitations in simulating large populations of neuron using biologically-realistic neural modelling. Some attempts in the past to implement these simulations in hardware, for better performance, are also described. The chapter ends with a discussion of the need to design an Application Specific Integrated Circuit (ASIC) computing system specifically for the purpose of supporting large-scale neural simulations.

3.2 Neural Network Simulation

For a correct computer simulation of any real object, we need to capture the object's structure, its functionality, and the environment affecting its behaviour. For an aircraft simulation, we simulate its functionality with a mathematical model, its structure is defined by design parameters, while the environment is generated by simulating the effect of various outside factors affecting the flight in isolation or in any combination. Only a correct model can lead to a functionally

correct design. In the case of neural simulation, the results can only be correct if we are simulating neural structure, functionality and environment as realistically as possible. The neural functionality is simulated using mathematical models that capture a neuron's dynamics, the network structure is controlled using variables and constants, while the environment is controlled by the neural connectivity. It is then important to simulate this population in a realistic manner, incorporating internal and external factors affecting the population's behaviour, such as noise and external stimuli [Tra02]. A neural model based on assumed functionality or imaginary connectivity may lead to unrealistic results [Izh03b].

Often, a complex simulation that uses high-fidelity details may result in a more complex model than the simulation objectives require [Tra02]. The simulation cost can, however, be reduced through the use of abstract modelling techniques [MM00]. These techniques reduce the model complexity by eliminating, grouping, or estimating model variables at a less-detailed level without grossly affecting the simulation results. Key issues in the abstraction process involve identifying minor structural and functional details, the leaving out of which does not affect the intended simulation objectives. However, the level of abstraction must be chosen carefully as wrong assumptions may lead to wrong results. The level of abstraction depends on the scientific question being investigated. If we are interested in the development and verification of a hypothesis related to the ionic conductances and the resultant generation of action potential in neurons, a detailed cellular level simulation capturing the complexity of a single neuron would be required such as that defined by Hodgkin and Huxley [Izh03b]. However, the simulation of a single neuron may fail to exhibit a holistic view of the complex behaviour and information processing mechanisms in a large neural network [Tra02]. The understanding of networks of interacting neurons is an area of major interest in many disciplines as there is not enough understanding of such a "non-linear interacting system" [Tra02]. Large neural networks have information processing capabilities, as a whole, beyond those of single neurons, such as representing information in a distributed manner. Contrary to a human-made artifact such as an aircraft, where the structure of each component with its functionality is known, a neural processing model is not understood in detail at the individual neuron and group level. Various mathematical models have been proposed to capture the functionality of neurons at various level of detail. Izhikevich [Izh03b] gives a detailed account of all of these models with their biological plausibility and

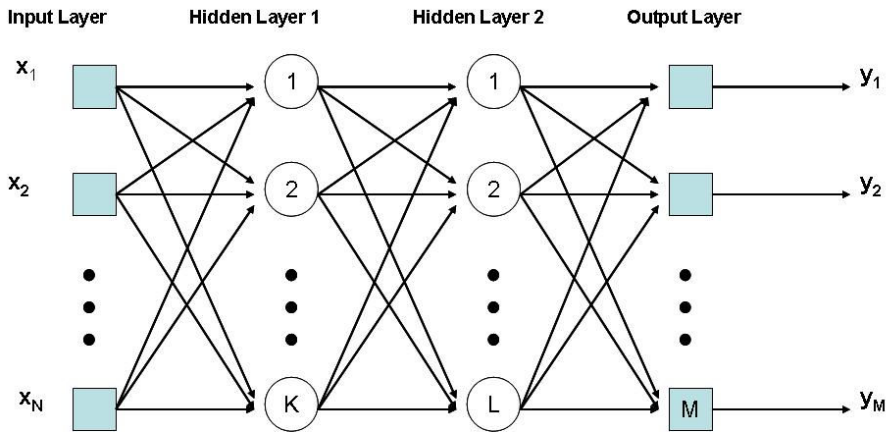

Figure 3.2: Traditional Multilayer Neural Network.

the computational power required to simulate them. One might wish to model neurons with their all known biological features, however, such an implementation may be too computationally expensive to simulate large neural network with the available resources.

A traditional neural network computer application such as the Light Efficient Network Simulator (lens) [Roh99], used in AI and psychology for training neural networks for various applications, uses rate-coded models. Rate-coded neural networks use the 'mean firing rate' (over time, over several repetitions, or over a population of neurons) as a means of information representation [Me98]. The neuron adds the weighted input values to compute the output using some mathematical function (such as the McCulloch-Pitts or sigmoid function [Tra02]. Traditional multilayer neural networks are based on a number of neuron layers including an input and output layer along with a number of intermediate hidden layers as shown in Figure 3.2. Each layer consists of an arbitrary number of neurons. The input layer provides the initial input state of the network, while each neuron in the middle layer accumulates weighted inputs from the previous or the same layer to give its output to the neurons in the same or the next layer. For supervised learning, the difference (delta) of the output from the required output is back-propagated to all connected neurons in the previous layers to adjust their connection weights and reduce the output errors. Many types of neural network such as feed-forward, simple recurrent, backpropagation through time, and fully recurrent etc., can be implemented with these simulators [Roh99]. These simulators model basic functions of neurons such as spiking (sending data among

neurons), weighted connections, functionality to accumulate the inputs and fire outputs. However, they lack the ability to simulate the correct behaviour of biological neurons as they do not capture the correct temporal spiking dynamics of individual neurons as stated in Chapter 2. In these models, the average firing rate for a single neuron is well defined under stationary conditions or over a period long enough to achieve an output. However, none of these conditions resembles those of biological neurons in most of their temporal behaviours [Me98].

Over the last few years the emphasis of research in neural network simulation has been towards pulse-coupled spiking neuron models for biological realism. The proposed models range from very detailed but computationally expensive models, such as that from Hodgkin and Huxley [Tra02], to the very simple Integrate-and-Fire (I&F) neural model. A detailed analysis of these models has been presented by Izhikevich [Izh03b] along with their capability to exhibit most of the prominent biological features and their computational complexity. Neuroscientists feel comfortable with the models that capture details down to ion channels, such as that from Hodgkin and Huxley, while computer scientists are happy to simulate simple models such as the I&F model. Each one's preference is a matter of concern to the other group as the Hodgkin and Huxley model [HH52] is computationally very expensive while the I&F is too naive to be biologically accurate. An efficient and realistic simulation warrants a compromise between the two extremes. The Izhikevich model [Izh03a] is one such compromise that exhibits most of the behaviours expected from biological neurons [Izh03b]. The mathematical model from Izhikevich that captures the dynamics of spiking neurons [Izh03b] is as follows:

$$v^{'} = 0.04v^2 + 5v + 140 - u + I \tag{3.1}$$

$$u^{'} = a(bv - u) \tag{3.2}$$

with after-spike resetting as:

$$if \quad v \geq +30mV, \quad then \begin{cases} v \longleftarrow c \\ u \longleftarrow u + d \end{cases} \tag{3.3}$$

where v is the membrane potential of the neuron, u is the membrane recovery variable which takes care of K^+ activation and Na^+ inactivation currents and provides negative feedback to v. After the potential reaches a threshold ($+30mV$), the membrane potential and the recovery variable are reset as per Equation (3.3)

above. $0.04v^2 + 5v + 140$ is chosen empirically to scale v to mV and time to ms [Izh03b].

The model claims to exhibit all known firing pattern of cortical neurons by suitable choice of its parameters a, b, c, and d [Izh03a]. The model is computationally comparable with the I&F model as it takes only 13 floating-point operations to simulate 1 ms (compared to 4 floating-point operations for the simplest form of I&F). It is efficient compared to the Hodgkin and Huxley model [HH52] which takes 1200 floating-point operations to simulate 1 ms of biological time [Izh03b]. The model is suitable for the simulation of large-scale neural networks as shown by Izhikevich [EMIE04, IE08].

In the following sections, various approaches to neural simulation will be discussed to highlight the importance of creating an efficient simulation engine as a motivation for the SpiNNaker research project. Though typical rate-coded multilayer perceptron (MLP) neural network simulations have made great contributions to research in understanding the brain and neural applications, to conserve space this chapter focuses only on neural simulations using biologically-inspired models.

3.3 Expected Features

Neural simulations are performed with various objectives such as understanding the behaviour of an arbitrary size of neural population (a neural network) to investigate various aspects of neural dynamics or learning. These may also be directed towards understanding the working of a particular part of the brain based on data obtained from biological investigations such as fMRI. Moreover, the objective may also be to test hypotheses relating to the functionality of an individual neuron or a group of neurons, or to investigate the information processing taking place in a particular portion of the brain as a result of known stimuli etc. Whatever the objective may be, in general, the neural simulations are expected to share the following expected features:

3.3.1 Biological Realism

To obtain reliable results, an experimental model must capture functionality as realistically as possible. Additionally, we must simulate the correct environment

to simulate valid behaviour of the model. A wrong assumption will produce inaccurate results leading to unrealistic inferences. As discussed earlier, neural applications based on unrealistic neural models delivered results that were far from reality. Reliable results from a neural simulation can be obtained only if the mathematical model is based on realistic and exhaustive study of biologically-realistic functional data. For this purpose a modeller needs to have enough knowledge or have a close interaction with biological researchers to capture the true functionality of the neuron. Often, a particular neural model may successfully be able to depict an accurate behaviour of a single neuron, however, using the same model may fail to generate an accurate behaviour of a population of neurons due to many factors such as spike communication noise [Tra02] or non determinism in the spiking probability of the neurons under certain conditions. In this case, the model needs to be revised with the help of biological statistics to ensure that it exhibits appropriate collective behaviour.

3.3.2 Scale of Simulation

For a meaningful functional behaviour of a biologically-plausible neural network at the scale of the functional units in the mammalian brain, such as the visual or auditory system, we need to simulate a sizable population of neurons [Tra02]. A reasonable size of neural network to simulate such functions in a mammalian brain is of the order of 100,000 neurons [EMIE04, IE08]. Medical scientists often need a simulation to encompass a specific region of the brain to understand the effects of stimuli and to verify expected responses. This requires a simulation of a network consisting a population in the order of millions of neurons. In order to simulate the social behaviour of mammals, we may be required to simulate a basic mammalian brain such as that of a rat containing $2x10^9$ neurons.

3.3.3 Simulation Time

As described in Chapter 2, neural dynamics is characterised by its temporal behaviour on millisecond time-scale [Tho00, Izh03b, Tra02]. Ideally, for a simulation with real-time stimulus and response behaviour, we should simulate neural networks in real time. However, for a large-scale neural simulation to the scale of millions or billions of neurons with biological realism, we require huge computing resources or it will not be possible to simulate the neural population behaviour

in real biological time. We can achieve this either by simulating a small population of neurons with limited functionality, or we need a very powerful computing system.

3.3.4 Abstraction Level

For any computer-based simulation, we need to abstract the the details of the modelled artifact for efficient simulation on the underlying computing system. Neuroscience can provide the details of a neuron's behaviour down to molecular-level, ionic-level, or cellular-level. Depending on the aim of the simulation, we can ignore the details unnecessary for the behaviour we aim to simulate. If we are interested in a behaviour-level simulation of a large population of neurons, ionic- and molecular-level activities in each neuron may not matter; thus they can be abstracted away. The abstraction level chosen, however, depends on the desired objective and may require the use of a different model for each such objective.

3.3.5 Interactive Control

In an aircraft simulation we need to have some control over the factors affecting the behaviour of the aircraft in order to view their individual or combined effects. In the same way, a neural simulation should provide support for user interaction to manipulate factors affecting the behaviour of the network and to analyse it accurately. The behaviour of mathematical neural models can be controlled by adjusting their parameters. A neural simulation should provide means to control such parameters externally. Similarly, the user may wish to view the state of simulation at a particular point in time for diagnostic purposes to examine the effect of a particular stimulus or the effect of particular neural activity over a period of time.

3.4 Software Neural Simulations

Computer Engineers have long been looking for a computationally-efficient model to emulate a reasonable population of spiking neurons. In the past, with limited processing power and memory resources, the Leaky-Integrate-and-Fire(LIF) model was used most extensively for research. Even now, this model is still used extensively and evolved further because of its computational simplicity. However,

as stated above, the research communities in computer science and neuroscience have are sceptical about its biological basis. Recent discoveries have forced researchers to look against biologically-inspired mathematical models of spiking neurons such as that from Hodgkin and Huxley [HH52]. Much research has focused on an intermediate solution to provide models with biological accuracy but not at the cost of computational efficiency [Izh03b]. Izhikevich's spiking neural model [Izh03a], which has been empirically derived from the Hodgkin and Huxley model [Izh07], is a result of one such effort that exhibits most known behaviours from many types of biological neuron and has reasonably low computational complexity [Izh03b]. In the following sections example software-based simulations are explored, and their limitations are described.

3.4.1 Example Software Simulations

Numerous simulations of spiking neural networks have been carried out in the past with various objectives employing various neural models. However, there are very few examples of large-scale neural simulations using biologically-realistic models. Here we present a few such examples to evaluate software-based approaches to large-scale neural network simulations.

E.D. Lumer *et al.* created a large-scale computer model in 1997 to study the synchronous rhythms in the thalmocortical system simulating 65,000 spiking neurons and 5 million connections from the visual sensory region [LET97a, LET97b]. The model used the I&F spiking neuron model to exhibit the neural dynamics of individual neurons. The simulation was driven with external stimuli and the neural state for all the neurons was recorded. The model simulated a 0.5 ms time step in 5 seconds on a Sun SPARC-20 workstation. The work claims to simulate the neural responses to visual input and the resultant synchronous oscillations, similar to the fast rhythms recorded *in vivo*. By systematically modifying physiological and structural parameters in the model, specific network properties were found to play major roles in the generation of this rhythmic activity. In another simulation performed with the same computer model, Lumer et al. studied the normal behaviour of the model during visual stimulation for the effects of disrupting synchrony by introducing a random jitter of a few milliseconds in the timing of action potentials.

Izhikevich performed a simulation of a biologically-inspired neural network consisting of 100,000 neurons with about 8.5 million synaptic connections in

2004 in order to study the effect of axonal conductance delays and Spike-Time-Dependent-Plasticity (STDP) on the formation of neural groups [EMIE04]. The Izhikevich-Simple-Neural-Dynamics-Model [Izh03a] was used to depict the neural dynamics of the neurons while the learning process (synaptic weights adjustment) was based on the STDP model proposed by Markram [MLFS97]. The simulation provided valuable results to help understand the behaviour of large-scale neural populations; which is not possible with a single neuron simulation. In another experiment performed by Izhikevich with G.M. Edelman in 2008, a population of 1 million spiking neurons was simulated with about half a billion synaptic connections [IE08]. The simulation aimed at understanding spontaneous activities, sensitivity of the group's behaviour to the changes in the individual neuron's responses and the functional connectivity in the mammalian brain by simulating multiple cortical regions and the connections among them. The density of neurons and synapses was reduced by a factor of 4 to reduce computational complexity. The initialization data was acquired from Diffusion Tensor Imaging (DTI) and fMRI scans of human, cat and rat brains.

The famous Blue Brain project [Mar06] was started in 2005, as a joint venture between EPFL Switzerland and IBM, with a view to reverse-engineering the mammalian brain, to investigate brain function and dysfunction through a detailed neural simulation. In its first phase, the project aimed to build a cellular-level model of a 2-week-old rat somatosensory neocortex corresponding to the dimensions of a Neo-Cortical Column (NCC) adopting biologically-accurate Hodgkin Huxley ion channel models for individual neurons. An NCC model comprising 10,000 neurons with trillions of possible connections was simulated employing the huge computational power of an IBM Blue Gene(L) supercomputer with up to 8192 processors. Currently, the time required to simulate the circuit is about two orders of magnitude larger than biological time simulated. The Blue Brain team is working to streamline the computation so that the circuit can function in real biological time [Swi08].

3.4.2 Limitations of Software Simulation

Software simulations are very flexible and reusable as changes can easily be incorporated, and further simulations can be built using the modules from previous attempts. However, as the simulation performance is dependent on the underlying hardware platform, the scale of neural simulation and the simulation time

depend on the computational power which is limited in case of a general-purpose computing system. The performance is further reduced by the overheads introduced by the operating system and other applications running at the same time. As a result, we need either to reduce the size of the simulation or to abstract the functional details. Either approach affects the speed of simulation and usually a large-scale simulation with reasonable functional details runs much slower than the simulated neural networks, as noticed in the above examples. Simulation time is not important for rate-coded-models as these networks are not normally characterised by their temporal behaviour. However, for a spiking neural simulation, the information processing depends on the spike timing which is very important for the neural dynamics and learning behaviour. As a result, biologically-inspired models with fine level of details may take considerably larger than the real time to simulate a short duration of neural activity.

As stated in the last section, the neural network simulation model by E.D. Lumer *et al.* simulated 0.5 ms of activity for a population of 65,000 neurons in 5 seconds on a Sun SPARC-20 workstation using the simplest spiking neuron model. Similarly, the first simulation by Izhikevich[EMIE04] took about 60 seconds on a 1-GHz Pentium PC to simulate 1 second of simulated neural activity with 100,000 neurons using the Izhikevich 'simple' spiking neuron model. The same was true for his second simulation with 1 million neurons that took 1 minute to simulate one second of neural activity in the brain on a 60 (3-GHz) PC cluster. In Blue Brain simulation, they could simulate a NCC comprising 10,000 neurons with an improved version of the biologically-accurate Hodgkin and Huxley [HH52] model on Blue Gene (L) supercomputer, the fastest machine in the world at the time of simulation, taking double the time the brain takes.

The computer science research community now seems convinced that the use of dedicated hardware is necessary in order to simulate large-scale neural network in biological real time, such as the Field-Programmable Gate Arrays (FPGAs) or Application Specific Integrated Circuits (ASICs). The best way to build such hardware is, however, not yet determined.

3.5 Hardware Neural Simulations

Some notably successful efforts have been made in the past to 'grow' neural networks in hardware, employing both analogue and digital circuits, for better

performance. The 'Neuromorphic Chip' by Indiveri [GID06] at the University of Salento, Italy, the "Distributed and Locally Reprogrammable Address-Event Receiver" by Alan Murray's group in Edinburgh [BMW08] and the "Configurable Wafer-Scale Hardware System" to model large-scale biological neural networks by Fieres *et. al* at Ruprecht-Karls University, Heidelberg Germany [FSM08] are some examples of analogue circuit design. However, all of these examples are on an experimental basis to grow only small-scale neural populations in order to establish the feasibility of neural modelling using analogue signalling techniques for inter-neuron communication.

On the digital circuit side, Tuffy *et. al* [TMM+06] uses time-multiplexing circuitry to facilitate inter-neuron communication to simulate neurons. However, the neural application is hard-wired with the achievable on-chip densities limited to about 3000 neurons/chip [RYKF08] due to the use of a bus architecture for spike communication. Reconfigurable FPGA architectures, e.g. [HAM07], are another popular technique. Here the entire network is multiplexed by swapping out physical components. FPGAs are also popular for component abstraction e.g. Porrmann's implementation of neural networks on a reconfigurable hardware accelerator [PWKR02]. The component abstraction technique reduces size by developing modules with generic functionality, which can implement arbitrary neural function (often with some simplification). FPGAs, however, are notorious for high power consumption, slow speed, and cumbersome reconfiguration. Application-specific hardware embedding dedicated abstract neural components, such that by Eickhoff [EKR06], is emerging as an alternative solution. These devices can be very general-purpose but still suffer from routing overheads unless combined with time-multiplexing techniques. The advantage of hardware simulation, common to all these examples, is the simulation speed that matches or exceeds the biological real time.

3.5.1 Remaining Challenges

The implementation of spiking neural networks on specially-designed hardware for accelerating neural simulation has encouraged the research community to adopt biologically-inspired spiking neuron models for neural simulation. Some experimental success stories demonstrate the efficient modelling of neural networks in hardware. However, these efforts lack some of the expected features of an effective neural simulation described in Section 3.3. Almost all of these efforts with

hardware implementation simulate small-scale neural populations much below the scale of a functional unit in the mammalian brain. No hardware implementation, to our knowledge, uses a biologically-realistic model which displays the major behavioural characteristics of spiking neurons (the I&F model, which is not biologically plausible [Izh03b], is mostly used in these implementations). One more problem is the rigid implementation approach i.e. the hardware is designed with a fixed spiking neural model embedded into the design which cannot be changed at run-time. If we want to refine the model or use a different neural model, this is not possible with these implementations. The brain has many classes of neurons which display differing behaviours. A particular neural model may not capture every kind of neuron's behaviour, or at least a different set of parameters is required to exhibit different behaviours. Techniques described previously do not offer this facility to model various types of neurons or to change parameters at run-time. Lastly, these models operate at hardware speed with no relation to the real-time neural activity in the brain, i.e. either these are faster or slower than the simulated biological neural network. We run a model and then try to establish a temporal relationship with the *in vivo* nervous system. If we want to use these hardware platforms to interact with some artificial artifact such as a hominoid robot, we require additional hardware to incorporate real-time delays in stimuli and responses travelling between the simulation and the robot.

3.6 Summary

Computer-based simulation is a very effective tool for investigating the behaviour of an existing or under-design artifact. Almost all engineering and physical science disciplines use simulation to understand complex phenomena and use the knowledge acquired to build new applications or improve the existing artifacts. Computer simulations also help in situations where repeated experiments with actual subjects may not be possible due to health-and-safety or social and ethical regulations such as those related to medical science. The human brain is a masterpiece of nature's engineering characterized by its performance, fault-tolerance, power consumption and heat dissipation. A balanced combination of these features is yet to be achieved by human engineering. This calls for scientists and engineers to learn from biology to explore the underlying principles and techniques employed by nature which need to be incorporated in man-made artifacts.

Computer-based simulations of large-scale neural networks help us to understand the functional and structural behaviour of the mammalian brain. A realistic simulation to provide reliable results requires simulation of a large-scale neural network using biologically-plausible mathematical models with a reasonable degree of detail commensurate with the desired results. Simulating large-scale spiking neural networks in real-time requires a computing platform with significant computing resources. Software-based simulations with their inherent overheads over a general-purpose computing system cannot reach this objective. Neural networks grown on FPGAs are limited in their scale and suffer from some other problems. To simulate the brain, we need to alter our engineering paradigm to conform to the biological engineering concepts. Biology believes in simplicity, parallelism, redundancy and slow processing at the micro-level to generate high-performance and fault-tolerant functional units at the macro-level. There is a need for an ASIC designed neural network simulation engine (i.e. a "brainbox") to simulate large-scale spiking neural networks in biological real time so that the functional mysteries of nature's engineering can be understood.

Chapter 4

The SpiNNaker Computing System

4.1 Introduction

The SpiNNaker research project at the University of Manchester started in 2006 with an aim to provide a high-performance computing platform for large-scale neural simulations in biological real time. The previous chapter identified a need for a specially-designed hardware platform to support large-scale spiking neural network simulation in real time. This chapter introduces the SpiNNaker computing system (known as the "brainbox"), which is designed to model a part of the mammalian brain to the scale of a functional unit in the neocortex; it is an Application Specific Integrated Circuit (ASIC) platform for large-scale spiking neural simulation. This chapter highlights SpiNNaker's objectives, architecture, certain important features from the developer's point of view, and an envisaged standard application model to support large-scale neural simulations. Finally a few important guidelines to develop applications for SpiNNaker are given along with some users' expectations acquired from our interaction with potential users of the SpiNNaker computing system.

4.2 SpiNNaker Objectives

The SpiNNaker computing system has been designed to support large-scale neural simulation using highly-parallel distributed computing with high-bandwidth inter-process communication. The envisaged computing system comprises up to

1 million low-power embedded processing cores interconnected by an efficient packet-switching asynchronous network. Inspired by biology, the system is being designed with following objectives in mind:

- High-performance
- Scalability
- Fault-tolerance
- Low power consumption

.

The aim is to enable the simulation of over 1-billion spiking neurons using a biologically-inspired neural dynamics model of simple spiking neurons such as that from Izhikevich [Izh03a]. With its 1 millions processors running at 200MHz, we are expecting a throughput of over 200 tera-instructions per second. The processors are organized in small Chip-Multiprocessor (CMP) Systems-on-Chip (SoC) for better scalability and fault-tolerance. A system of the desired scale can be assembled by connecting the required number of CMPs. Circuit boards with varying numbers of interconnected CMPs will be available to be connected together to form computing systems with varying throughput. Design effort has concentrated on producing a reliable system with fault-tolerance at both component and system level. Despite the scale of the system, we try to keep the system as 'green' as possible by using low-power embedded processing cores and a power-efficient network to connect them. The energy-efficiency per instruction is far from being close to that of the mammalian brain. However, it is much lower than typical massively-parallel computing systems of this scale. Besides its low-power hardware design, we propose a power-efficient application model for the SpiNNaker computing system that further reduces the power consumption by removing software overheads and keeping its embedded processing cores in sleep mode for most of the time.

4.3 Architectural Overview

The SpiNNaker computing system can be considered as a large cluster of multi-core, independently-functional, personal computers (PCs), where each PC can run a number of homogeneous or heterogeneous processes to execute an overall

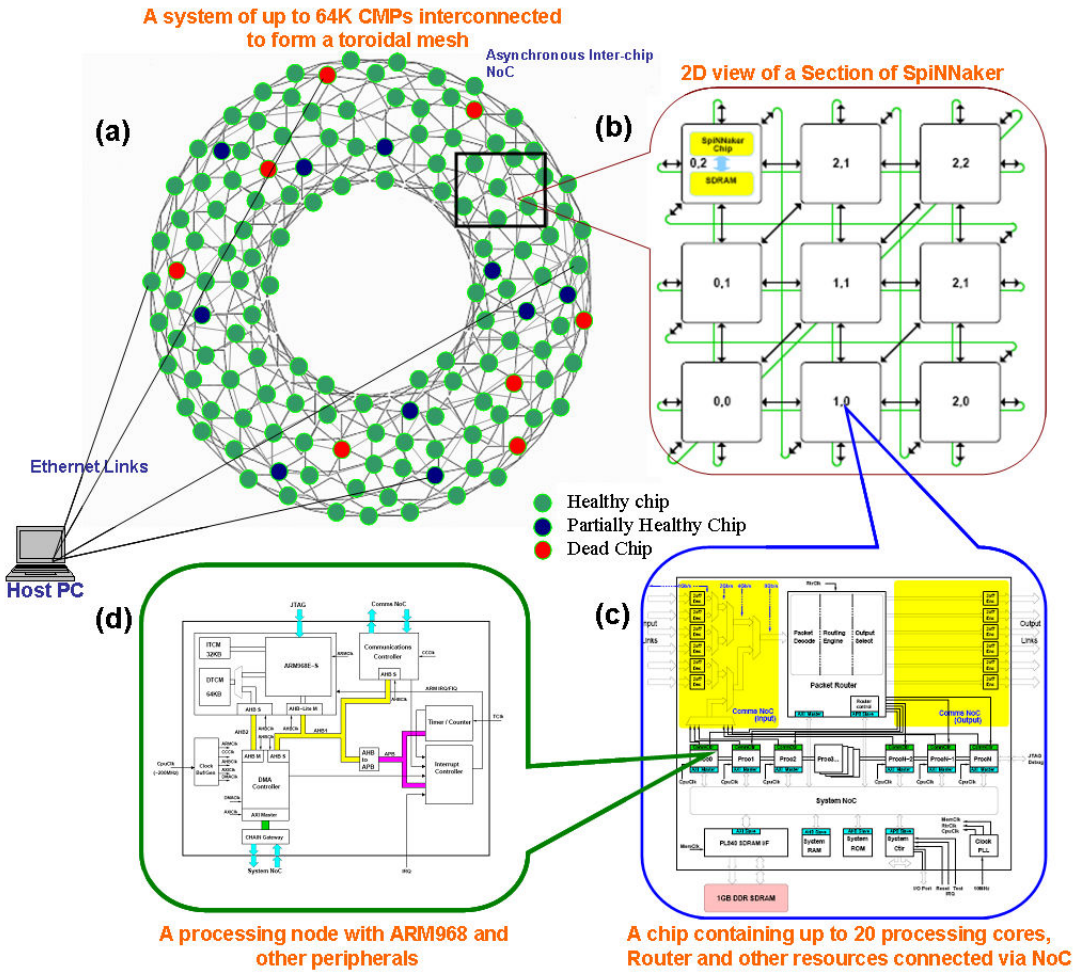

Figure 4.1: SpiNNaker Computing System [Pro07].

system-wide parallel distributed application. In a cluster, the processing nodes communicate by passing inter-process messages or through a shared memory. Each process runs independently and its state depends on the initial state, the information received from the other processes over time, and the processing performed on the information received. In SpiNNaker, the user interacts with the system and the application with the help of a PC connected to the SpiNNaker system, called the "Host PC". The Host PC communicates with the system through Ethernet links connecting one or a few chips on the system as shown in Figure 4.1 (a). The SpiNNaker system may comprise varying numbers of SpiNNaker-CMPs interconnected via their six bidirectional links as shown in the Figure 4.1 (b) to form a toroidal mesh. The SpiNNaker-CMP (Figure 4.1 (c)) contains a router to join the external links and to connect the 20 on-board processing nodes with the help of an asynchronous packet-switching network. Each processing node (Figure 4.1 (d) is itself an independently functional unit with its own resources to support neural computation. Along with the processing nodes, each CMP contains chip-level shared resources such as memory. A detailed description of these components is given in the succeeding paragraphs. The system is divided into three levels of hierarchy: at system-level SpiNNaker behaves like a cluster of PCs, at CMP-level it is a multicore PC along with memory and networking resources, while at the processing-node level it is like a PC less its mass memory and networking infrastructure. The succeeding sections give a bottom-up description of the SpiNNaker computing system at various levels.

4.3.1 SpiNNaker Processing Node

The SpiNNaker computing system uses low-power embedded ARM processors to execute neural processing code. We have chosen ARM968E-S from the ARM9E family for its high instruction throughput, low power consumption and small area [Ltd08a]. Each processing node (Figure 4.2) is formed around the ARM968E-S processing core with its dedicated Tightly Coupled Memories (TCMs) comprising a 32-Kbyte instruction memory and a 64-Kbyte data memory. Running at 200 MHz, the ARM968 can model over 1000 simple spiking neurons in real time, using models such as that of Izhikevich. The processor's private memory is sufficient to contain both code and neural state data for over 1000 simple spiking neurons. To support the neural processing, each ARM core is provided with supporting

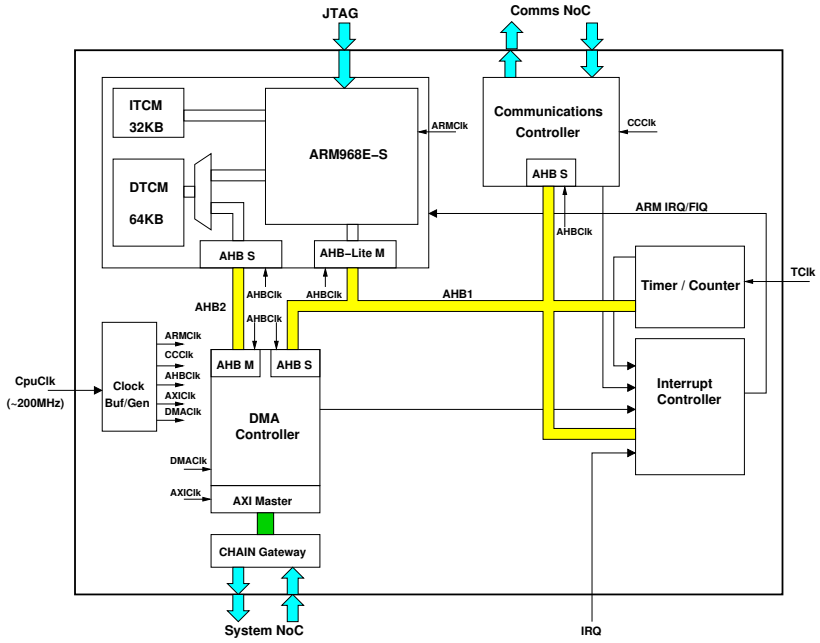

Figure 4.2: SpiNNaker Processing Node [Pro07].

peripherals including a Timer, an Interrupt Controller, a Communication Controller and a DMA Controller connected via an AMBA High-performance Bus (AHB). The Timer notifies the processor each millisecond to help perform the neural processing in real-time. We use the ARM Vector Interrupt Controller (VIC) which provides 16 vectored interrupts so that the processor can directly read the address of the related interrupt handling code from the VIC's vector address register. This reduces the execution time required to handle the interrupts and improves performance. The Communication Controller communicates with the asynchronous Network-on-Chip (NoC) connecting the processing nodes. It helps form packets before sending these to the other processors and decomposes them on arrival. The DMA controller provides a bridge to access the chip's resources over its asynchronous NoC. The DMA efficiently transfers data to/from the local memory from/to the chip shared memory while the processor is busy in its computation.

4.3.2 SpiNNaker CMP

Each SpiNNaker CMP (Figure 4.3) may contain up to 20 processing nodes each of which may implement a neural process to simulate a fascicle (a group of neurons with associated inputs) of spiking neurons. Each chip is provided with additional resources such as the System RAM, Boot ROM, System Controller, Ethernet Controller and a Multicast Router (MCRouter). These components are connected to the processing cores *via* an asynchronous Network-on-Chip (NoC) called the System NoC. The System Controller is a collection of important control registers used for chip-level configuration and management. It maintains the state of all chip resources. The Ethernet Controller can connect any chip to the Host PC while the MCRouter routes packets among processing nodes on the same or on other connected chips using another asynchronous network called the Communication NoC.

The processors have sufficient local memory to hold the neural dynamics code and the neural state information. However, this memory is inadequate for the synaptic information related to the simulated neurons. If 4 bytes are used to represent the information for each synapse (i.e. the weight and axonal conductance delay associated with that synapse), we require a minimum of (1000x1000x4 bytes) 4-Mbyte for each processor and hence a minimum of 80-Mbyte is required for the CMP with 20 on-board processors. To satisfy this, an off-chip SDRAM of 1-Gbit is provided to the synaptic information. The SDRAM is shared among the 20 processing cores and transfers information from/to the processors' TCMs as required, using the DMA Controller. SDRAM has been kept off-chip to support future upgrading of memory to a larger size. The SDRAM is connected to the processing cores through the DMA Controller *via* the System NoC which servers as a high-bandwidth shared medium among the 20 processing cores [RYKF08]. Both the NoCs are based on Silistix Ltd's CHAIN (CHip Area Interconnect) technology developed at the University of Manchester [PFT$^+$07] and provide a bandwidth of 1-Gbit/s at a much reduced power consumption; such a throughput is not possible with most typical bus architectures. Each chip has six bidirectional asynchronous links to connect the on-chip MCRouter with those on six neighbouring chips.

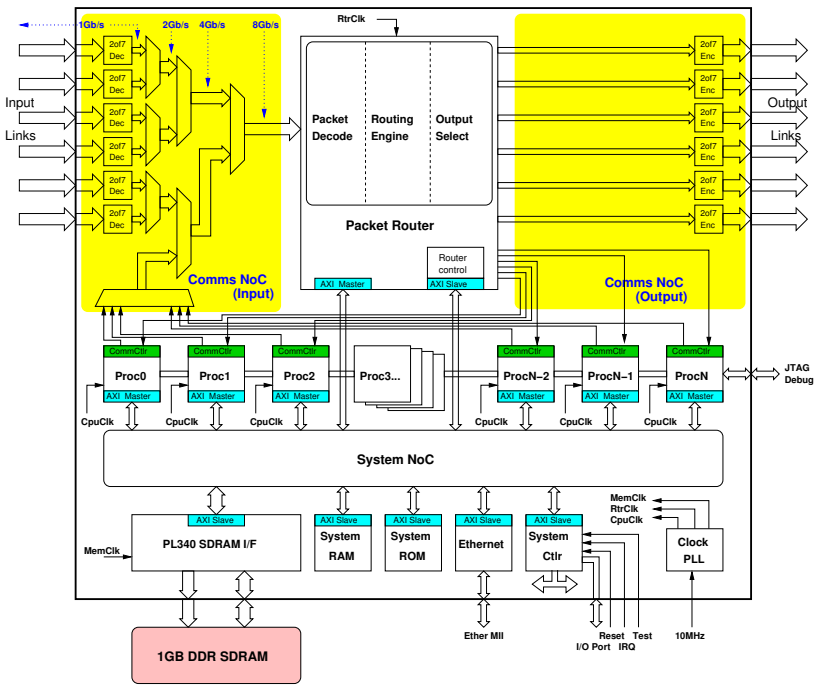

Figure 4.3: SpiNNaker CMP [Pro07].

4.3.3 Inter-Neuron Communication

Neurons communicate by sending spikes (action potentials) to each other. In spiking neural networks the only information a spike carries is the time of its firing and its source (the neuron it comes from). While the neurons are simulated on the processing nodes, the inter-neuron spike transmission between the processing nodes is supported as small (40- or 72-bits) packets over an asynchronous packet-switching network called "Communication Network" (Figure 4.4) connecting the on-chip processing nodes with the on-chip router, and then directly connecting the chips together through their six links. The hub of this network in each chip is a specially-designed on-chip router (the MCRouter) that routes the packets (spikes) to 20 internal outputs corresponding to on-chip processing nodes and to the 6 outward links connecting to other chips. The MCRouter can multicast a packet to any combination of the internal processors and the external links as a result of source-based associative routing. The multicast feature of the MCRouter helps simulate high spike fanout as the neuron's axons may be connected to other neurons located in the processing nodes on the same or other chips. The

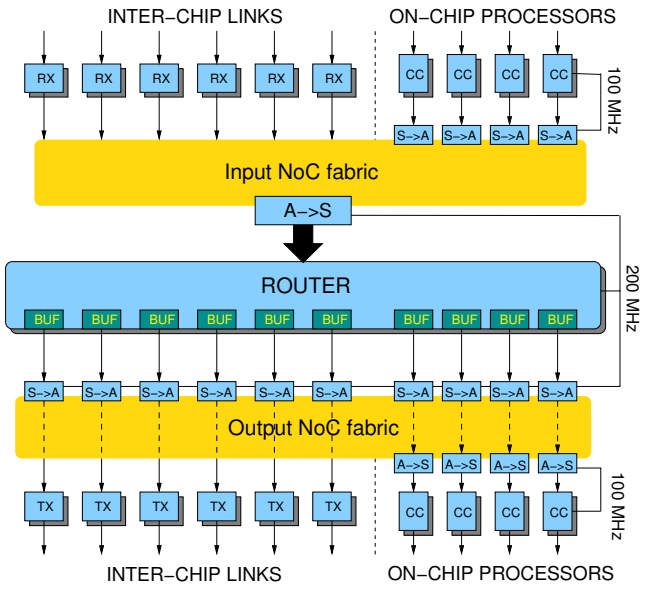

Figure 4.4: Spike Communication Network [PBF+08]

packets from six neighbouring chips and 20 on-chip processing nodes are serialized by the on-chip part of the Communication Network before passing these to the MCRouter. A system of any desired scale can then be formed by linking chips to each other with the help of their six bidirectional links, continuing this process until the system wraps around to form a toroidal mesh of interconnected chips as shown in Figure 4.10.

The Communication Network supports three types of packet: Multicast (MC), Point-to-Point (P2P) and Nearest-Neighbour (NN). MC packets are used to support spike communication among the neurons in each processing core, P2P and NN packets are used mainly for system management and diagnostic/configuration purposes. Figure 4.5 shows the composition of these packets. Each packet contains a 32-bit routing key which provides information to the MCRouter. The routing key is a 32-bit source address in the case of a MC packet, 16-bit source and 16-bit destination addresses in the case of a P2P packet, and a 32-bit instruction or memory address in case of an NN packet. The packet also contains an 8-bit control field to specify the type of packet, certain control fields to support the router in packet handling, and a parity bit for error detection during packet transmission. Besides this, a packet may contain an optional 32-bit payload to transfer data. The MC and P2P packets are routed with the help of routing

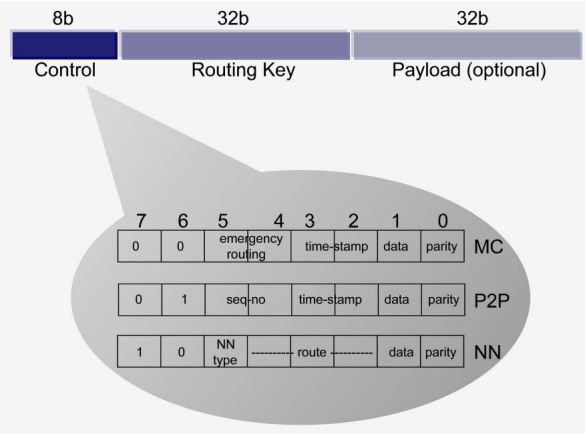

Figure 4.5: Packet Format [Pro07].

tables which require configuring before use, while the NN packet routing is hard-wired into the router and thus does not require any configuration. For multicast neural communication, the MCRouter contains 1-K words of associative memory as a look-up table to find a multicast route associated with the incoming packet's routing key. P2P packets are routed using destination chip-address with the help of a routing table with 64-K routing entries in each router, and is used by the monitor processors for system-level management. The MC and P2P packets contain a timestamp in their control. The MCRouter maintains a record of the timephase which is inserted in all the locally generated packets and is used to delete off-chip packets arriving from other chips that are two phases old. This feature helps remove free wandering invalid packets from the network [Pro07].

The MCRouter is an efficient hardware component which can route one packet per cycle at 200 MHz with the help of its six stage pipeline, multicasting it simultaneously to a subset of its 26 outputs (20 on-chip processors and 6 out-going links). The Communication Network supports a bandwidth of up to 8-Gbps per chip [PFT+07]. The following are some of the main features of the SpiNNaker inter-processor Communication Network:

- **Default Routing**: If no match is found, the router passes the packet to the link diagonally opposite to the incoming one. This process is called 'default routing' (Figure 4.6) and it helps to reduce the number of entries in the look-up table as with this technique we only require router entries at the source, destination, and on those chips where the packet needs to change direction. The global packet-switching network to support spike communication has

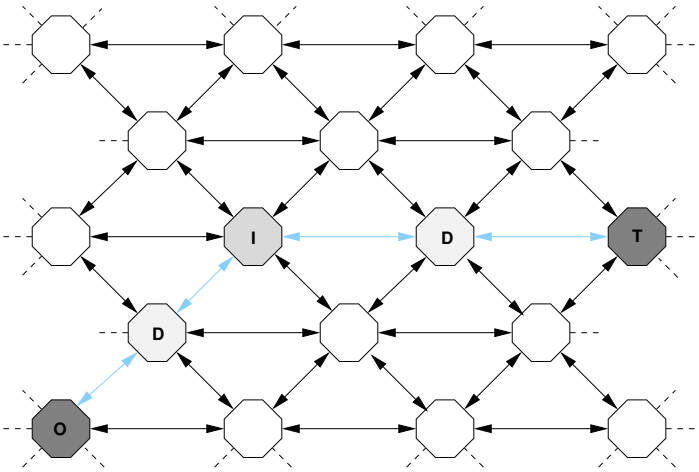

Figure 4.6: Multicast Routing – Default Routing.

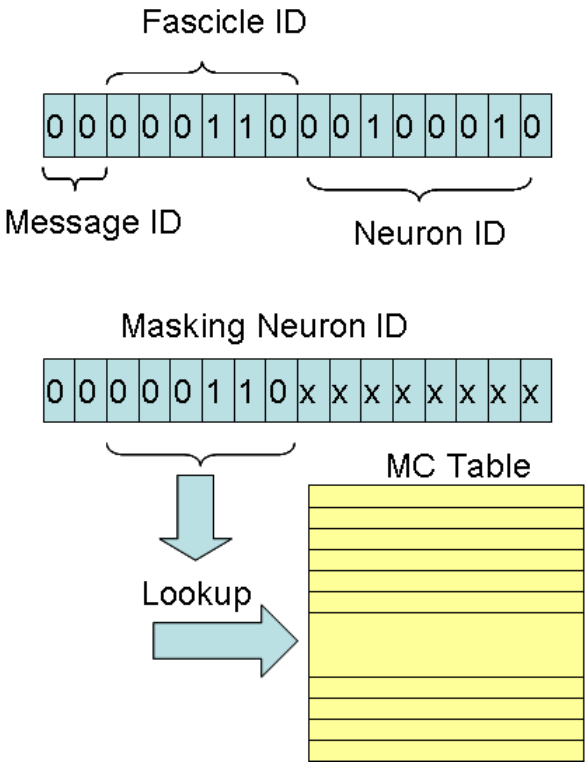

Figure 4.7: Multicast Routing – Masking the Bits.

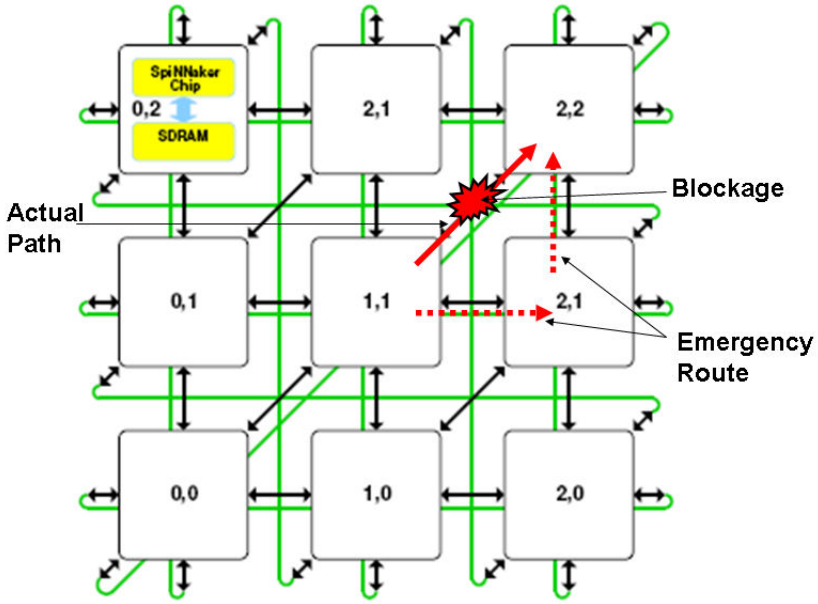

Figure 4.8: Multicast Routing – Emergency Routing.

been hierarchically organized to keep the address space to a manageable scale. On the global network, only chip addresses are visible. Each chip maintains a chip-level private subnet of 20 processing nodes which is visible only to the local router, while an individual neuron's identifier is local to the processing node. With this scheme we can keep the global address space to a limited number of entries in the routing tables. The router masks the bits containing the individual neuron identifier to look up only against the chip address and fascicle identifier as shown in Figure 4.7. The fascicle identifier is included in this lookup to identify packets destined for the local chip. This way, for a fascicle size of 256 neurons we can mask 8 bits carrying the neuron's identifier and thus can reduce the number of entries in the routing table to 64 instead of 16,384 ($= 2^{14}$) with a source identifier of 14 bits as shown in Figure 4.7.

- **Emergency Routing**: To deal with transient congestion at the outer links, a packet can be routed to its adjacent link as a measure of "emergency routing" as shown in Figure 4.8. The neighbouring chip's router then returns the packet to its correct path.

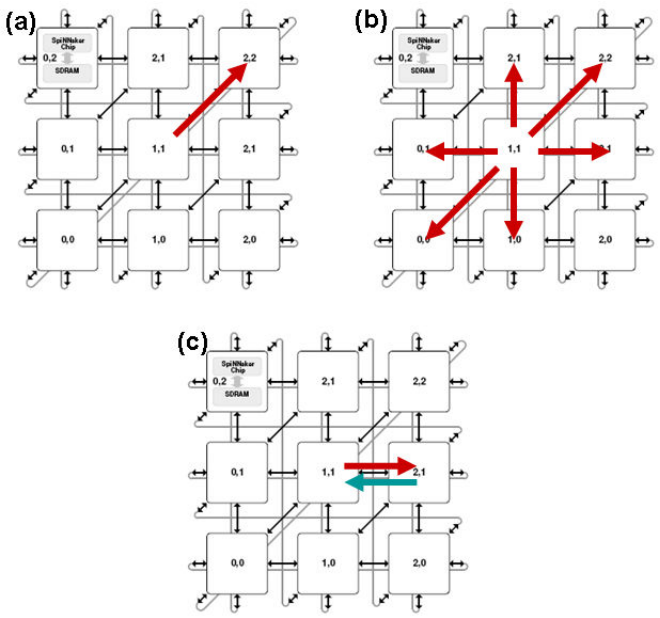

Figure 4.9: NN Packet Routing (a) To a particular neighbour (b) Broadcast to all six neighbours (c) Peak and poke.

- **NN Packet Routing**: The NN type of packet is used for system configuration and diagnostic purposes. The packet can only travel among the nearest neighbours and does not require any configuration at the router. The SpiNNaker CMP can send an NN packet to any particular neighbour (Figure 4.9(a)) or to all the six neighbours at one time (NN broadcast)(Figure 4.9(b)). The NN packet can also be used by a chip to read from the chip resources of any of its six neighbouring chips or to write to any of its writeable location (peak and poke)(Figure 4.9(c)).

4.3.4 System-level Behaviour

The SpiNNaker computing system is connected to the Host PC by linking any one (or more) chip(s) through an on-chip Ethernet link as shown in Figure 4.10. Each chip is provided with an Ethernet Interface which can be connected to the Host PC with the help of a Physical Layer Module (PHY). However, in the presence of an existing efficient inter-chip network, only one chip or a few of these are connected to the Host PC and the Communication Network can be used for communication between chips. SpiNNaker has been designed as a "Universal Spiking Neural Network Architecture" to support a variety of neural

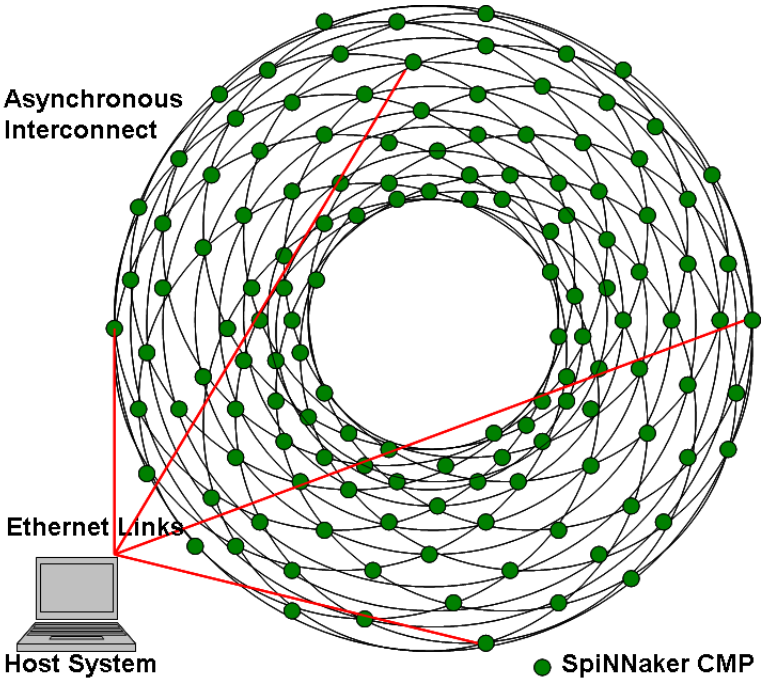

Figure 4.10: SpiNNaker System - a conceptual view.

network models. To this end, no neural dynamics model has been preloaded during manufacture. The system is designed to be configured for the required neural application at run-time. The Host PC provides the SpiNNaker system's interaction with the outside world and the same can be used for passing stimuli and responses to the application at run-time. The Host PC is also intended to support system debugging and to provide user-level interaction with the system or the application running on the SpiNNaker system.

4.4 Main Features

The following are the main feature of the SpiNNaker massively-parallel computing system:

1. **Parallel Distributed Processing**. SpiNNaker provides real-time parallelism where a neural process simulating a number of neurons runs independently in a separate application (fascicle) processor. As each process has a separate image in the processor's TCM, we can run heterogeneous neural

dynamics models concurrently on various processing nodes. This is an important feature which allows the simulation of heterogeneous neural models in the context of a large neural network application with many types of neurons performing various types of functions concurrently. Various jobs can be distributed among the on-board processors in the SpiNNaker CMP, such as a processor running an operating system to manage the chip resources and communicate with the Host PC for system level managerial jobs while a group of other processors are simulating neural dynamics to update the neurons' state. Yet another processor may be looking after the adaptive connectivity as a result of Hebbian-learning performed by the application processors.

2. **Dynamically Configurable Network**. The spike communication network is organised for inter-neuron connectivity by configuring the routing tables at run time. Thus, the Communication Network is very flexible and can support any type of connectivity in a neural network. Any neuron in any processing core can communicate with any other neuron in the system with a few micro seconds transmission delay over the Communication Network. This feature allows neurons to be placed virtually anywhere in the system, though physically co-located in the nervous system. We propose the co-location of neurons to form fascicles (a group of neurons with related inputs) on the same processing core or on the same chip (in case of large size fascicles) to reduce network congestion [KLP+08]. The toroidal form of the network provides multiple paths between any two neurons, and the routing tables can be configured for the optimal path (with respect to time or distance as the two may be different due to traffic conditions). The routing tables can be reconfigured at run-time by one of the processors as a result of learning or to handle congestion. The ability to reconfigure the network at run-time helps fault-tolerance at the system-level.

3. **Large Distributed Memory**. The chip has three levels of memory: fast local memory with each processing core, on-chip shared System RAM, and an off-chip relatively large (1-Gbit) SDRAM. The processing core's local memory is used for critical data such as the neural dynamics code and neural state data for the 1000 simulated neurons. Each processor has sufficient local instruction memory (32-Kbyte) to hold real-time application code such

as the Interrupt Service Routines and the neural dynamics code, while the data memory (64-KB) is sufficient to hold the neural state information and a lookup table to point to the relevant synaptic data from SDRAM as and when required. Off-chip SDRAM is used to hold data not required urgently, such as the synaptic data for the 20,000 neurons being simulated on a chip which is only required selectively when a neuron receives a spike. The on-chip System RAM is relatively faster than the SDRAM and is used mainly for inter-processor message passing and by one of the processors for the chip-level management, such as keeping recovery routines, chip status, chip-level global variables, and an image of the microkernal (if used). The DMA controller in each processing node localizes the view of the data in the SDRAM by seamlessly transferring data in and out of the local memory on demand.

4. **Connectivity to the Outside World**. The system can communicate with the outside world by connecting its chip to the Host PC using on-chip Ethernet interface. This link can be used to give a system-wide view to the user on the Host PC and is intended to be used for the system configuration and loading the neural application into the system at run-time. It can also be used to communicate with the application at run-time. Run-time system-level management and fault-handling can also be performed with the help of this link.

5. **Scalability**. The SpiNNaker computing system has been implemented as CMPs. This allows a system of the desired throughput to be built with up to 64xK chips. Even a system with only one SpiNNaker-CMP can function independently to simulate a population of up to 20,000 neurons with their connectivity defined in the MCRouter which can route packets among the on-chip processors. PCBs with varying numbers of SpiNNaker CMPs (16x16, 32x32 etc.) can be made available to assemble computing systems of the desired scale.

6. **Low Power**. The SpiNNaker-CMP uses the processing cores from the "smallest and lowest power" ARM9E family [Ltd08a]. The processing nodes are designed specifically for running a neural simulation as an embedded real-time application. The ARM968E-S is chosen for its efficient and sufficient TCM to run a neural simulation for a group of spiking neurons. The

processor has efficient low latency bus interfaces (an AHB-Lite master- and slave-interface) for the DMA to reduce power consumption and improve responsiveness [Ltd08a]. The processor has dual-banked TCM data memory providing full bandwidth access by the core and DMA. The SpiNNaker standard application model, explained in Section 4.5, also helps to preserve energy by keeping the processor in a sleep state whenever there is no useful work to be done.

7. **Fault-tolerance**. The SpiNNaker hardware supports error detection for several types of chip-component faults, and fault handling through configuration and management software. The SpiNNaker system provides redundant resources at each level of its design to support fault-tolerance. The SpiNNaker fault-tolerance features will be discussed later in Chapter 8.

4.5 The SpiNNaker Application Model

The SpiNNaker CMP system has been designed to support spiking neuron simulations in real time. One of the ways in which spiking neuron models are considered different from typical rate-coded models is in their temporal behaviour [Me98]. This means that the model simulating a neuron has to exhibit temporal properties very close to those of a biological neuron. As explained in Chapter 2, neurons can be considered as event-sensitive cells which keep changing their state, i.e. their membrane potential, over their life span; this activity is in millisecond domain [Tho00, Tra02, Me98]. Each neuron can receive input spikes on its dendrites (in the order of 1000) and send an output spike through its axon (to multiple dendrites of other neurons). These input and output events cause the neuron state (the membrane potential) to change on a millisecond time-scale. From a real-time neural modelling point of view, the modelled neurons should display accurate neural state at a millisecond granularity with correct input/output spike activity. This model is quite different from a typical Multi-layer Perception (MLP) rate-coded neural network simulation model [Roh99], where all neurons in the network update themselves on synchronized 'ticks' of a clock which does not follow any real-time temporal logic.

In SpiNNaker a fascicle (group) of neurons is simulated by a neural process running on an ARM968 processing core. This process is responsible for updating each neuron's state at 1 millisecond intervals based on the mathematical model

employed to exhibit neural dynamics for the simulated neurons. This process is also responsible for accumulating the stimuli (synaptic weights) on receipt of spikes during each update interval. Spike transmission is realized with the help of multicast packets over the Communication Network. The 'spike-received' event is initiated by incoming packets to a processing node. The neural process identifies the target neuron(s) in its fascicle with the help of the source Identifier (ID) received as part of the routing key. Each processor maintains a connectivity lookup table for this purpose. For each target neuron, the process adds the synaptic weight associated with the synapse between the source and target neuron to the accumulated stimulus for the target neuron during that update interval. We abstract the analogue behaviour of a neuron by continuously updating its state with millisecond granularity as a function of the accumulated stimulus received for the duration of the last millisecond. As a result of this update, some neurons' state may warrant firing a spike. A spike is sent as an MC packet containing a routing key composed of the source ID of the spiking neuron, the fascicle ID and chip-address. The routing tables in each chip are configured to map the simulated neural network's inter-neuron connectivity onto the SpiNNaker hardware. The MCRouter in each chip directs the spike packets to the destination neurons on either the local processing nodes or in the direction of the chip(s) containing those neurons.

In biological neural networks, spike transmission is characterised by axonal delays of some milliseconds (1 ms to 40 ms [Tho00]) as the spike travels with a maximum speed not more than a few metres per second. However, the spike-packets in the SpiNNaker system travel at electronic speed taking a few microseconds. For realistic temporal behaviour, we need to present the spikes to the neurons only at their exact real time i.e. a spike received now should be delayed by its associated axonal delay before being accumulated into the neuron's stimulus [RKJ$^+$09]. We maintain time-based bins for each neuron to accumulate stimuli until this projected update time [JFW08] as shown in Figure 4.11. The axonal delay is stored along with the synaptic weight associated with each connection in the SDRAM, while the lookup table that points to this information is maintained in the data TCM. On receipt of a spike, the neural process looks up the corresponding connection information against the routing key (the source neuron's ID) in the received packet. On receipt of this information, the synaptic weight is accumulated into each connected neuron's stimulus bin corresponding

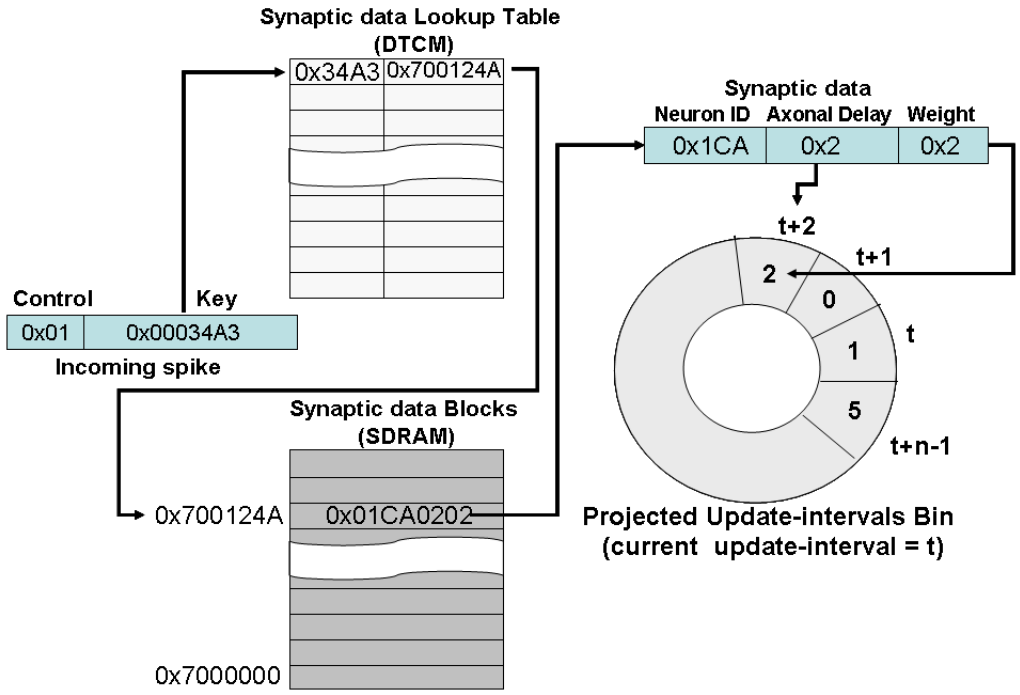

Figure 4.11: SpiNNaker Standard Application Model - Stimulus Update Process on Receipt of a Spike.

to the axonal delay on that connection [JFW08].

As explained in Section 4.3.2, synaptic information is stored in the SDRAM. With this separation of data from the processing core, on the receipt of a spike the neural process must retrieve relevant synaptic information into the local memory before accumulating the synaptic weight into the projected stimulus bin as shown in Figure 4.11. As fetching relevant data from SDRAM introduces a delay during which the processor may receive more spikes, the job of fetching data from SDRAM is assigned to the DMA Controller. The data retrieval time is still insignificant relative to the axonal delay and does not incur too much overhead to update the state of all simulated neurons. Completion of data retrieval is indicated to the processor with a 'DMA-completion' event, upon which the neural process adds the weight to the relevant stimulus bin in the local memory as shown in Figure 4.11.

This process continues in real-time to maintain an up-to-date neural state of every neuron in the simulated neural network. As the neurons do not fire more than once in a millisecond and at any time normally only 0.1-1% of the neurons

are firing, we expect each processing core to remain idle for a considerable amount of its simulation time. To conserve energy, we put the processor to sleep during its idle time.

4.6 Hardware Support

These three events, i.e. the spike-received event, the update-interval (millisecond) event, and the DMA-completion event govern the functionality of the neural dynamics to maintain the neural state for each neuron in the fascicle processor. The events are signalled by hardware interrupts from the processing node's peripherals to its Interrupt Controller. The Communication Controller generates packet-received (spike) interrupts, the Timer is configured to provide an interrupt to the Interrupt Controller after every update interval (millisecond), while the third interrupt is generated by the DMA controller to indicate DMA-completion. The Vectored Interrupt Controller (VIC) in each processing node can be configured to present the address of the relevant Interrupt Service Routine (ISR) for each interrupt to the processor. This saves processing time as the processor does not have to poll to find the source of the interrupt. The VIC can also be configured to prioritise nested interrupt handling by the processor. The neural dynamics process is implemented with the help of the ISRs associated with these interrupts. The ISR for the packet-received interrupt reads the packet from the Communication Controller, uses the lookup table to find the address in SDRAM containing the relevant synaptic information, and requests a DMA operation to fetch the relevant data from the SDRAM. The ISR associated with the DMA-completion interrupt accumulates the synaptic weights for each connected neuron in the stimulus bin corresponding to the associated axonal delay. The ISR for the Timer interrupt calls a function to update the neural states of all neurons in the fascicle processor. Figure 4.12 shows the standard application model with the ISRs along with their proposed priorities.

With this application model, we run the whole application with the help of ISRs in a manner akin to an embedded real-time event-driven application. This provides an efficient model of execution without the need for a scheduler at the processing nodes. The application runs under the control of the VIC which prioritises the scheduling of the ISRs as hardware threads running as part of a neural process on each application processor. Besides performance, the system

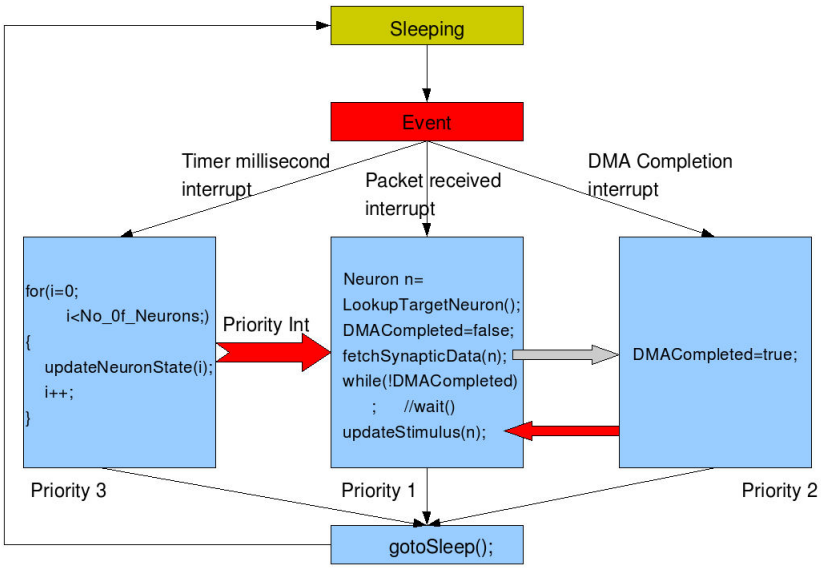

Figure 4.12: SpiNNaker Evnet-driven Application Model with the Help of ISRs.

conserves energy by putting the processors into the sleep mode after every ISR.

4.7 Development constraints

The following are some of the constraints to be kept in mind while developing an application for SpiNNaker:

- **Memory Usage**. As described earlier, the SpiNNaker system has a lot of memory. However, it is distributed over the whole system. The neural simulation code is run from the processor's local instruction memory and the neuron state data is kept in the local data memory which is very limited in size (32-Kbyte instruction and 64-Kbyte data TCM). This will impose a limitation on the choice of spiking neural model, the implementation technique, or the number of neurons to be simulated in each processor. This implies that the application developer cannot use the memory without regard to its size; the critical code and data must be kept within the local memory limits, while other data is kept outside the local memory. As there is no hardware memory management unit, scheduler, or paging software in the application processors, the application developer needs to incorporate

these features in his code if he wishes to use the external memory for his application. A large application can be run from the SDRAM, but the performance may not meet the real-time requirement.

- **Routing Table Size**. There are only 1000 (32-bit) entries in the multicast routing table. The neural connectivity in the nervous system is based on locality and grouping; most connections are to neurons in close vicinity and neurons are grouped with related inputs [EMIE04]. The number of table entries seems insufficient. However, if we configure the system intelligently [KLP+08], these entries are sufficient to simulate a large-scale spiking neural network. We propose grouping neurons with related inputs in the same processor to require very few entries in the routing tables. Besides this, if we use the multicast feature of the router to send a packet towards many output ports at the same time, many routes can be combined to utilise only one routing entry. As a guide a router should have an entry for the packets originating that chip, destined to that chip, or changing direction in this chip en route. Other packets take the default route as explained in Section 4.3.3. It is important that the application is configured meticulously for optimal neural mapping to keep the router entries within the limit.

- **Efficient Code**. To simulate the behaviour of a real-time neural network, the neural process in each processor needs to update the neurons' state in each millisecond. During this time, the processor may receive a number of spikes along with other interrupts such as DMA completion, shared memory message passing or error detection interrupts from its peripherals. Neural simulation functions are performed by ISRs, which prohibit any lower priority interrupts during their execution. To meet update interval bounds of 1 millisecond and to enable the processor to process all interrupts, these functions must be coded efficiently or a delayed response to an important interrupt may have unwanted effects on the overall application.

- **Synaptic Mapping**. An SDRAM of up to 1-Gbit has been provided with each SpiNNaker-CMP to hold the synaptic information associated with each dendritic link to the on-chip neurons. Approximately 4 bytes are required to hold the synaptic weight and axonal delay with the index of the neuron in the chip. If a spike is targeted to all (1000) simulated neurons, we need

to bring in 4-Kbytes of data from the SDRAM on each spike receipt and then accumulate the stimuli for all the neurons before the next millisecond interrupt. As all 20 on-chip processors are performing the same kind of activity, the SDRAM access may slow down the process. An efficient scheme is required to make the process efficient. One such technique is to divide the SDRAM into 20 segments, one for each processor, to minimise contention for data access. Similarly, creating a lookup table in the data TCM can reduce the time required to bring the data into the local memory.

- **Debugging Support**. SpiNNaker application developers and users will wish to see simulation results in some readable form for comparison with expected results. Unlike a normal PC application, the debugging and diagnostics of a large-scale massively-parallel computing system running a real-time application are not simple. In a real-time system the state changes continuously, whereas for diagnostic/debugging purposes we need to have a stable application state at some point in time. This is difficult to achieve in a continuously running system.

- **Synchronization**. The system as a whole is distributed with no centralized clock. However, for system-wide behaviour we need a synchronized notion of time for certain application requirements such as stimulus and response, passing application state, or timestamping the application state. This is a difficult problem to handle and can only be done by defining an acceptable time skew, such as a few microseconds, with the help of synchronizing broadcast packets. The application developer needs to keep this constraint in mind, or devise some way to achieve application-level synchronization with the help of packets.

- **Event-Based Model**. Any type of neural network, from a typical rate-coded MLP to a biologically-inspired spiking neural model, can be implemented on the SpiNNaker computing system. However, the hardware has been designed specifically to support an even-driven application model. There are many ways a neural application can be implemented on SpiNNaker, however, for efficient use of the hardware and to gain maximum performance, we propose using the SpiNNaker application model for application development (as explained in Section 4.5).

4.8 User Expectations

The majority of researchers working with neural network applications still use rate-coded MLP neural networks. In the World Congress on Computer Intelligence (WCCI2008), of over 1600 papers presented on neural simulations or applications, only a few used spiking neural networks (mostly using primitive Integrate-and-Fire spiking neuron model). This reflects the amount of work being done in the area of MLP neural networks. Spiking neural networks are not yet a well-explored area of research and transforming the work done with MLPs to spiking neurons is not an easy task. From another viewpoint, neuroscientists may not be happy with computer scientists' levels of abstraction, which may obscure "minor" details "vital" to them such as the ion channels, ion concentration, dendritic compartments potentiation, and slow chemical messaging etc. Another problem is the effort required to map an existing spiking neural network onto SpiNNaker or change the already-developed applications to conform to the SpiNNaker application model. Some concerns which emerged during discussions with the potential users are summarised here:

- Researchers wishing to carry on working with MLP neural networks, such as those from psychology and artificial intelligence, might want to use SpiNNaker for its high computational power to run large-scale neural network applications. However, they won't wish to change their applications to conform to the spiking neural network models. This requires altering the SpiNNaker application model to suit rate-coded neural networks. It poses a challenge for the system engineers and application developers in the SpiNNaker project to tailor the SpiNNaker execution model to support sigmoid neural networks or to run these applications as event-driven applications without the notion of real time.

- Some neuroscientists may wish to simulate very detailed models of spiking neurons to visualise the effects of some organic or chemical factors on the neuron's behaviour. A detailed neural model such as a refined form of the Hodgkin and Huxley model would be their choice, with inherent computational cost. Such models can be implemented on SpiNNaker at a much reduced scale. Conversely, a class of researchers in computer science may wish to have a large-scale neural simulation to visualise the behaviour of a

large-scale neural population using very simple (Integrate-and-Fire) models. It is a challenge for the SpiNNaker system team to meet such a variety of researchers' demands.

- Developers of neural network applications on a PC or PC cluster may be concerned about the low-level details of the SpiNNaker computing system. The development constraints explained in Section 4.7 imply that the developers must write their code at a device-level using the ARM instructions set, optimised for speed and memory size. Developers familiar with high-level compilers and MPI over advanced operating systems which abstract the architectural details and give a single PC view of a cluster, may wish to have the same level of abstraction for the SpiNNaker system. They may wish to have a Hardware Abstraction Layer (HAL) to insulate them from the architectural intricacies of the SpiNNaker computing system and an API with low-level functions to automatically transform their applications written for a single PC or a PC cluster to the SpiNNaker architecture. They may also wish to have a user-friendly interface at the Host PC to help configure their applications for SpiNNaker, debug the application, and interactively communicate with it at run-time to obtain meaningful results.

4.9 Summary

The SpiNNaker massively-parallel computing system has been designed specifically to provide a high-performance and fault-tolerant hardware simulation engine for simulating large-scale neural networks in real-time. The system is built around CMP technology using simple low-power embedded processing cores along with the use of an asynchronous NoC to conserve energy and minimise heat dissipation. It is a scalable architecture where the CMPs can be connected together to form a system of the desired throughput over an efficient asynchronous interconnect. The system supports parallel distributed computing to enable the running of large applications with special emphasis on fault-tolerance. The standard application model proposes running an application as a real-time embedded application consuming low power. In this model, all processors remain in a sleep mode unless woken up by some event. The event-driven application is based on the ISRs called as a result of interrupts from the interrupt controller. The whole application is run through the ISRs which are responsible for performing specific

jobs before forcing the processor to sleep again. For an efficient application to simulate a real-time behaviour, an application developer needs to make use of the design features of the SpiNNaker computing system to obtain maximum use of its ASIC design. The design also poses some challenges to the SpiNNaker team to satisfy potential users already working with the MLP neural networks who wish to use SpiNNaker for running MLP applications with better performance. We need to provide them with some hardware abstraction layer to enable them to transform their current applications for running on the SpiNNaker computing system.

Chapter 5

System-Level Simulation

5.1 Introduction

Hardware components can be designed, synthesised and tested with the help of commercially available Computer Aided Design (CAD) tools, which can exactly simulate their behaviour in the design phase. However, component-level simulation may fail to exhibit a holistic view of a complete computing system's functionality. A system-level simulation models an entire system and is able to simulate application-level functionality. It can help in developing and testing applications for a hardware platform while the system is still in its design phase. These models provide a reasonably accurate picture of the system in terms of their functionality and performance. Besides this, software simulation of a full hardware system before its fabrication helps in studying architectural tradeoffs. This is especially true in the case of a System-on-Chip (SoC) which consists of many independently functional components including Intellectual Property (IP) modules. These systems are hard to build and debug, and require a rigorous systematic approach to design as debugging by trial and error can not be afforded in hardware manufacturing [Fur05]. Software modelling techniques must to be employed to verify an architecture for functional correctness during its design phase.

As part of this research, a complete system simulation of a multi-CMP SpiNNaker system has been developed for design verification and application development/testing. We adopted the SystemC Transaction Level Modelling (TLM) technique to model the SpiNNaker CMP with all its architectural details. In

TLM-based modelling we are not concerned with the accuracy at signal and register level but concentrate on an accurate behaviour at a broad functional level, simulating inter-component communication as atomic transactions. We can simulate a complete system at various levels of abstraction to achieve simulation performance depending on the level of detail required.

This chapter covers our motivation for creating a system-level model of the SpiNNaker computing system. The available choices for simulating a System-on-Chip (SoC) are discussed i.e. high-level (transaction-level) modelling vs. a typical Register Transfer Level (RTL) simulation. An overview of the SystemC TLM technique as a choice for high-level simulation is analysed. To conclude, we describe the verification performed to validate the model itself and its abilities to simulate an accurate behaviour at chip- and system-level, and the experiments performed with the help of this simulation to verify the design objectives of the SpiNNaker computing system.

5.2 Simulating a Complete Computing System

A complete system simulation captures an entire computing system at its full instruction set architecture level, allowing it to run intended applications unaltered [AM00], while providing an accurate timing model. These simulations allow the development and testing of target applications while the actual hardware is still in the design phase. As these simulations are software-based, they are fully deterministic in behaviour and can be tailored for specific uses, and allow a test-bench to be attached for analysis without interfering with the actual system's functionality. A system-level simulation also facilitate the temporal debugging of an entire target system along with its target application as a single package. This is important if the hardware is being designed for a specific application and we want to verify the design for its intended objectives. These simulations are especially helpful for testing real-time applications as the application and the underlying hardware can reliably be debugged simultaneously for real-time analysis. As the model is functionally identical to a real system the application can be run unmodified, limiting the sources of error to those introduced by the hardware. Besides functional correctness, a system-level model is usually reliable in predicting execution time at the desired temporal resolution. However, high-fidelity temporal accuracy jeopardizes simulation speed, so for the simulation of a large

computing system running a complex application, we need to trade temporal accuracy for speed by approximating low-level details. The degree of approximation depends on the simulation objectives [AM00].

5.2.1 Related Work

In the past, there has been a little research on complete system-level simulation for a multiprocessor computing system. Mostly, commercially-available simulators are used to test the functionality of an SoC-based hardware design. One such example is SimOS [Ros95, Her96], which originated as a PhD research project but was commercialized later to facilitate application testing. The simulator supports a number of commercially-available processors, memory blocks, controllers, DMA and other active and passive components. Initially designed to work as a single processor simulator, it was later upgraded to support multi-core parallel computing systems to a certain scale. However, the system does not support the simulation a very large-scale computing system (such as SpiNNaker), as the performance goes down (at least linearly) as we increase the number of processors [Her96]. The model is also not suitable for a distributed computing architecture (such as the one proposed for SpiNNaker), as all the processing cores use a single application image. Moreover, as SimOS has been designed to support most commonly-available commercial architectures, customizing it to conform to non-compatible changes requires a substantial implementation effort.

Two similar system-modelling tools are Augmint [AS96] and SimpleScalar [BA97]. Both simulators support multiprocessor systems based on the x86 architecture. These, however, are not complete computing system simulations and are primarily used for examining the functionality of one particular set of components at a time [ea03]. The complete system simulation for temporal debugging of general-purpose operating systems and workloads by Albertsson *et al.* [AM00] is another example of a complete system simulation based on the Simics simulator from Virtutech [Ltd08b]. They developed their own models for the components not available in the Virtutech Model Library and developed an interface to support temporal debugging of the application along with the hardware system. Simics [Ltd08b] supports multi-core simulations with a variety of processing cores and other peripherals. It is a very sophisticated tool with a graphical user-interface and application development/debugging tools to support a full-scale SoC hardware-software co-design at TLM level. However, it is a proprietary simulator

and requires licences for the components being used from its library.

IBM created a full-scale system simulation, BGLSim, for its Blue Gene/L computing system, acquiring many functionality features from SimOS to add to their already-available multiprocessor simulator called Mambo [ea03]. The simulation was run on a Linux cluster to simulate a complete massively-parallel computing system. A Linux kernel image was loaded into the IBM PPC processors' models to support running applications. A number of applications were run for collecting performance statistics, debugging new applications and validating design choices before building the actual hardware [ABBea03]. However, the model is proprietary to IBM, models IBM-based architectures, and uses proprietary libraries to implement the system.

5.2.2 SpiNNaker Complete System Model

SpiNNaker is a fully-distributed computing system with three levels of functional hierarchy. At the lowest-level, ARM processing cores along with their peripherals form independently functional processing nodes. At an intermediate-level, 20 such nodes are interconnected via an asynchronous Network-on-Chip (NoC) along with other chip components such as MCRouter, System RAM, ROM, SDRAM, Ethernet Controller etc. in the form of the SpiNNaker CMP. At the highest level, multiple SpiNNaker CMPs are connected using an asynchronous interconnect to form the SpiNNaker computing system. As the nodes at each hierarchical level are independent functional units which interact by sending and receiving packets over an asynchronous communication infrastructure, the system at one level up in hierarchy may generate a holistic behaviour quite different from its underlying components. The spike communication, for example, cannot be simulated with a single processing node. Similarly, the application loading mechanism is quite different in a multi-CMP system from one with a single chip connected to the Host PC. The former requires the transmission of the application received from the Host PC as Ethernet frames to the other chips using small packets on the asynchronous packet-based interconnect, while the latter receives these for its own use only and is quickly ready for execution. A neural simulation on a multi-CMP system requires more configuration than a single CMP application. For these reasons, a complete system-level simulation for the SpiNNaker computing system was considered important to understand and verify the system behaviour with single and multiple chips.

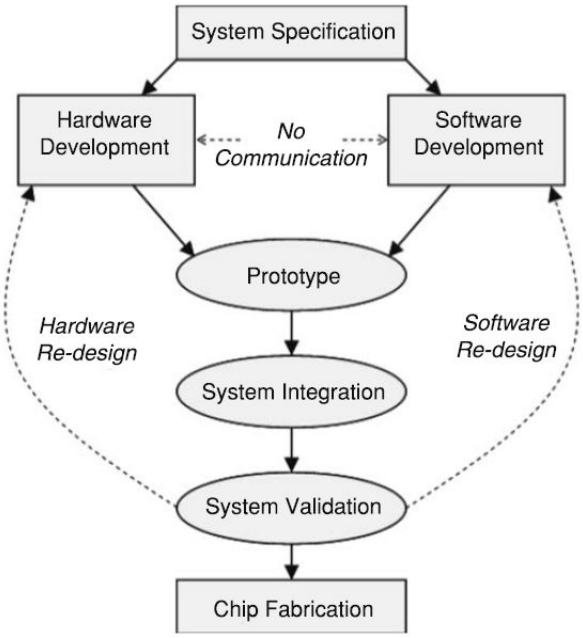

Figure 5.1: Classic Hardware Design Flow [Ghe05].

5.3 System-level Simulation Options

Hardware Description Languages (HDL), such as Verilog and VHDL, are useful for implementing specifications and evaluating various architectural trade-offs during the hardware design phase. These are an important tool for simulating component behaviour at Register-Transfer Level (RTL). Once a component has been simulated correctly, the code can be synthesized using physical libraries. However, these languages are not suitable for modelling large-scale systems as the high-fidelity component details become a performance bottleneck. Another disadvantage of using an HDL is its inability to simulate a model at various levels of abstraction as required in the initial phases of hardware design. A typical design flow for designing a piece of hardware using an HDL is shown in Figure 5.1.

High-level languages such as C++ can simulate a complex system architectures, such as an SoC, much faster than its HDL model. For example, booting Linux takes 30 seconds on a C++ based high-level simulator of an SoC design, 7 hours on a hardware-accelerated VHDL simulator and more than 20 days on an RTL model with standard VHDL [ABBea03]. With a high-level language, we

can capture various views of the design at various levels of abstraction. However, a high-level language may not capture an accurate architectural behaviour for lack of standard hardware related features readily available in the HDL. The model developed with C++ is of no use at later phases of the design cycle such as logical and physical compilation. Moreover, in the absence of a standard C++ modelling technique, it is difficult to reuse the component models and share them among developers. There is a need to have a system-level simulation using higher languages such as C++, but the modelling practice needs to be standardised to be shared among developers around the world. Moreover, some standard hardware oriented features may be acquired from the HDL which can then be used as standard practice by all users with a common backend simulator.

5.4 SystemC Transaction-Level Modelling

A solution to all the issues in the last section, is a blend of HDL constructs with a high-level language in the form of the SystemC library, which provides an ability to achieve clear levels of abstraction with high simulation speed. SystemC has introduced Transaction Level Modelling (TLM) at a functional level of abstraction, which has proved to be the key to the fairly fast acceptance for this methodology by the hardware industry and the research communities. There are no proprietary issues affecting a purchase decision for a costly design tool. SystemC supports modelling hardware and software together at multiple levels of abstraction [Pan01]. This improves design productivity through a more reliable design methodology within a shorter design time-frame to enable software development earlier in the SoC design flow i.e. hardware/software co-design. The advantage is an architectural exploration to analyse the potential effects at various abstraction levels, balancing the trade-offs between speed and accuracy [Ghe05]. TLM has introduced a new SoC design flow as shown in Figure 5.2.

5.5 Advantages of SystemC TLM

Over the last a few years, SystemC TLM has emerged as the most successful technique for system-level modelling. For SoC design, it has become a standard for early functional validation due to its reliable methodology which improves design productivity. The following sections give the main features of SystemC TLM

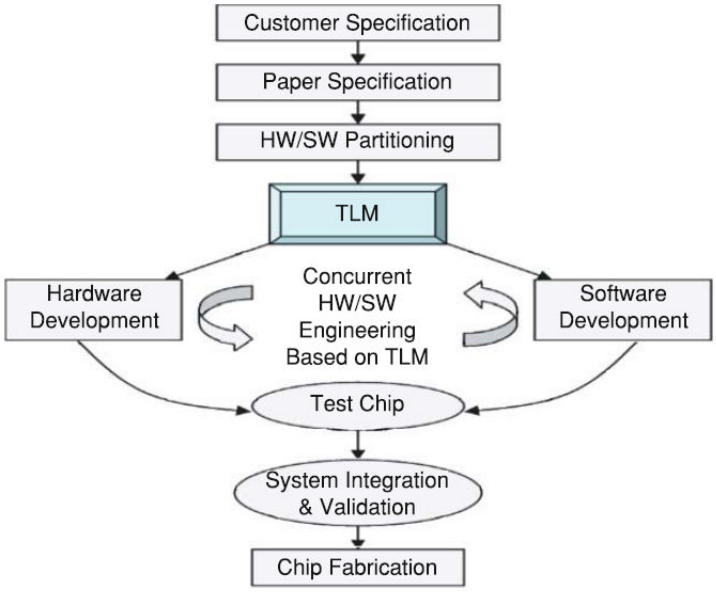

Figure 5.2: TLM Design Flow [Ghe05].

modelling [Ghe05] which make it a popular choice for system-level modelling.

5.5.1 Levels of Abstraction

In contrast to HDL-based simulation, SystemC TLM supports system simulation at various levels of abstraction. At the highest level of the design flow, it supports algorithmic modelling without detailed specification. As this captures only the design algorithm regardless of the implementation details, an algorithmic model has the advantage of having high simulation speed. However, at this level there is no notion of hardware or software components. The TLM Untimed Functional Model at Programmer's View (UTF-PV) is an example of such simulation. At the second level of abstraction, the model can be refined to include architectural details at a coarse level. The system comprises components communicating at transaction level without clocks, latency, or recovery time involved. This model, the Untimed Functional model with Architectural View (UTF-AV), is slower than the algorithmic model but provides greater detail. A Timed Functional Model at Cycle Approximation (TF-AV(CX)) and Cycle Accurate (TF-AV(CA)) levels are comparable with RTL for their timing behaviour and can be used for software development if an Instruction Set Simulator (ISS) model for the processing core

has been implemented. Even at this level of detail and accuracy, the TLM is many times faster than the RTL simulation and can be prepared in much less time than that required for RTL. SystemC also supports RTL modelling at signal and pin level, however, at the time of writing this thesis, the methodology is not mature enough to compete with standard HDL-based modelling.

5.5.2 Early Software Development

The most feasible way to develop software for target hardware is with the help of an HDL emulator or an FPGA prototype. However, the availability of these models is too close to the end of the hardware development to offer any time saving. The hardware design issues revealed by software testing at this stage will be too costly to fix. The TLM SoC platform can be developed on delivery of the system specifications. The target platform is available for software development much earlier in the SoC design cycle. Hardware and software development can start in parallel while the hardware is still in the design stage, the system can be verified functionally, and the hardware can be improved on the basis of feedback from software development. The TLM-based model can also be used as a "golden model" for verifying the RTL functional model as it is very close to the specifications.

5.5.3 Architecture Analysis

SystemC TLM offers an opportunity to explore a system's architecture shortly after the system specification is complete. An untimed functional model with only functional delays can be used to study the conceptual feasibility of the system while the timed TLM can be used for a thorough architectural analysis. A system optimization or modification can be tested in time- and cost-efficient way. Besides this, it can help improve design consistency between the hardware and software teams as both are working with the same functional model.

5.5.4 Functional Verification

TLM is the executable specification of a given design [Ghe05] that captures the intended behaviour perceived by the system designer. The software tests developed to evaluate the TLM can also be reused for functional verification of the

RTL model and the two results can be compared. TLM can thus save significant verification time.

5.5.5 Open Source Industry Standard

Hardware vendors have provided numerous methodologies and tools to support modelling tasks such as real-time performance estimation, executable specification, and hardware software co-simulation. However, they share little in modelling methodology. With these tools it is usually difficult to move from one abstraction layer to another due to the lack of any common interface strategy [HLWW02]. With the SystemC TLM IEEE standard [Com06] in place, all Electronic Design Automation (EDA) companies and Integrated Circuit (IC) suppliers comply with the same standard while implementing their SystemC class libraries which can be easily incorporated in all SystemC supported SoC design tools. Over a period of time, this has become an industry standard to be followed by all the hardware vendors for system-level modelling. The reference simulator along with standard TLM class libraries are available as open source public distributions at the Open SystemC Initiative (OSCI) and are supported by all known computer hardware vendors. Even the proprietary EDA tools conform to the same standard and a model designed with the help of the OSCI TLM library can be incorporated in any EDA tool supporting SystemC.

5.5.6 Real-time Debug Support

As the hardware and software are both simulated as one package in a high-level language, both can be debugged at the same time to understand real-time hardware-software interactions. This is important for verifying the design concept at an early stage of the system design as a feasibility study.

5.6 SpiNNaker System-Level Simulation

For a system-level simulation of the SpiNNaker computing system, we have adopted the SystemC approach defined by the OSCI TLM principles [Gro02, Ghe05]. The implementation details of the model are described in the following sub-sections.

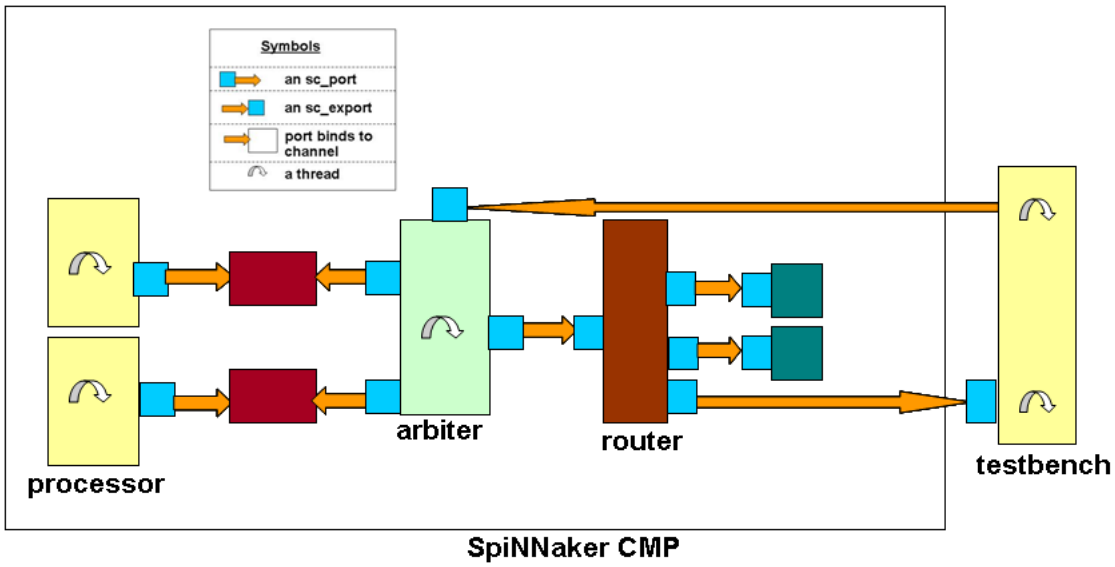

Figure 5.3: SpiNNaker CMP Model - UTF-PV.

5.6.1 SpiNNaker UTF-PV Model

As a first step in the system-level modelling, an Untimed Functional Model with a Programmers View (UTF-PV) [Joh06] was prepared without any architectural details (Figure 5.3). The purpose was to depict how a single-chip system will appear to a programmer at an abstract level. A test-bench was incorporated to test the functionality at application level. To test the packet-based communication, on-chip routing functionality was provided to route packets algorithmically at various ports of the chip connected to the testbench. The purpose was to examine the functionality of a SpiNNaker Chip-Multiprocessor (CMP) from a programmer's perspective focusing on how the on-chip processing nodes communicate with each other and with the off-chip nodes. Some basic chip-level functions, such as the support for sending and receiving packets, were visualized from the programmer's view and implemented at an abstract level as no architectural details were incorporated into the model. The model was based on the design specification from the initial SpiNNaker datasheet [Pro07]. The model used the channels from the standard OSCI SystemC library and the OSCI SystemC reference simulator was used to simulate the SpiNNaker CMP behaviour.

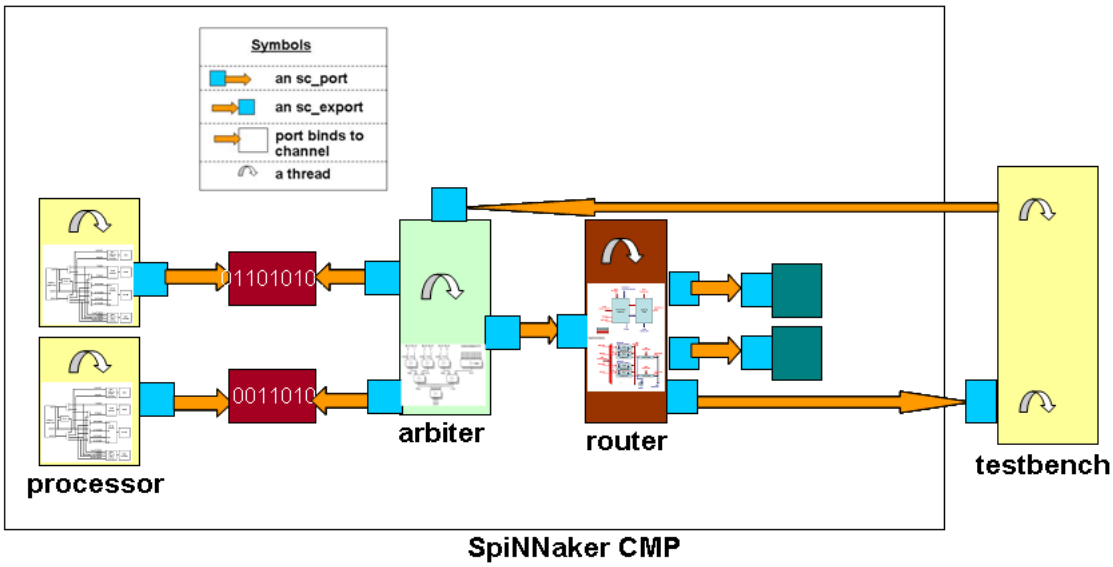

Figure 5.4: SpiNNaker CMP Model with Component-level Architectural Details - UTF-AV.

5.6.2 SpiNNaker UTF-AV Model

The SpiNNaker algorithmic model was refined in the second stage of its evolution into an Untimed Functional Model with Architectural View (UTF-AV)(Figure 5.4). In this stage, the model provided the architectural details as per the device specifications listed in the SpiNNaker datasheet. Complete functionality and architectural details were incorporated in various chip components such as the Communication Controller, Interrupt Controller, Timer, Router, Communication NoC, System NoC and memory blocks. The components whose functional specification was not cleared at that time, such as the System Controller, the TX/RX Inter-chip Interface, Watchdog Timer and the Ethernet Controller etc, were not included in the model. The details were taken from the datasheet before hardware design work had started on any of the components. A dynamic associative memory model was created to implement huge amount of SpiNNaker memory on a normal PC with its limited memory. The model used TLM bidirectional request-response channels from the SystemC TLM library to support the communication on the AMBA High-performance Bus (AHB) and System NoC, while TLM unidirectional channels were used for the Communication NoC. The model was then extended to multiple chips with logical connectivity that could be reconfigured automatically at run-time with the help of parameters in a configuration

file. The model was not fully untimed as we incorporated some estimated functional communication delays to acquire an estimated temporal behaviour of the application.

We conducted a case study to verify the system-level model in the first place, and then to understand the behaviour of SpiNNaker multi-chip system under a potential application workload. Details of this case study are described later. Based on our analysis some important design decisions were taken to meet new requirements exposed by the analysis, such as support for a Nearest Neighbour (NN) packet for diagnostics and debugging, the need for a System Controller with registers to support chip-level control functions etc. The design of Communication Controller was improved to include incoming and outgoing packet buffers and a dedicated register for incoming packets with some new interrupts to support the intended applications. In the same way, the MCRouter's Design was changed to include outgoing packet buffers and an error-handling mechanism. The need to have an Ethernet Controller on the SpiNNaker CMP to connect the system to the Host PC was realised and it was felt that the chip's external packet-receiving (Rx) interface should add the incoming port number before passing the packet to the router to assist the router in its routing decision, and then to pass this information to the processing node for application requirements. The most important contribution of this model was that it exhibited the multi-chip behaviour of the SpiNNaker computing system running a message passing application; this was a feasibility validation for the system design and gave us future milestones to be achieved with a very clear view of the actual system. At this point, we also started research into how to configure this system in the most dynamic way to support a variety of neural applications.

5.6.3 SpiNNaker TF-AV(CX) Model

The SpiNNaker UTF-AV model clarified many aspects of the SpiNNaker computing system. However, it lacked a real-time communication behaviour for studying how communication delays would impact the application. An intended application was developed with the ARM Instruction Set Simulator (ISS) on the ARMulator from ARM Ltd. [JFW08] which gave us a confidence in the SpiNNaker standard application model on the processing node. However, this work was confined to only one processing node's behaviour, simulating 1000 spiking neurons

using the Izhikevich spiking neuron model [Izh03a]. Some realistic communication behaviour was needed in a multi-chip system to analyse the feasibility of the SpiNNaker system. With this in mind, we transformed our SpiNNaker UTF-AV model into a cycle approximate Time Functional Model with Architectural View (TF-AV(AX)). The intention was to provide cycle-approximate behaviour of the system by incorporating the timing statistics obtained from HDL simulations and the commercial IP datasheets. We simulated all synchronous components at their specified clock speeds such as processors, AHB, Communication Controller, Router, Interrupt Controller and the memories. As the design work on some hardware components such as the Communication Controller and the Router had progressed to generate a behavioural simulation, we calibrated our model's functionality to match exactly their HDL simulations at cycle-level granularity.

A SystemC Intellectual Property (IP) model for the System NoC provided by Silistix Ltd. was incorporated into the model. As the IP was using communication interfaces from the Co-Ware Ltd. components library, these were made to communicate with TLM-based request-response channels with the help of "transactors" (modules to translate transactions between two different communication protocols [Joh06]). The processing core model was not implemented as an ISS as it was to be provided by ARM Ltd. In the absence of an ISS, the model could not run an application image developed to run on ARM968. To cope with the problem, the ARMulator was used to acquire the processing time, while the SpiNNaker TF-AV(CX) model provided communication delays corresponding to the multi-chip SpiNNaker application. The two results were then combined to produce an estimated overall application behaviour. As the application runs asynchronously in a parallel distributed manner over the SpiNNaker computing system with inter-process packet communication, this behaviour approximation is very close to the actual application behaviour (this was supported by our later cycle-accurate ISS based system-level model).

We conducted a case study to run a real-world application intended for the SpiNNaker computing system using the same methodology to validate the system performance; the case study will be described later. The SpiNNaker TF-AV(CX) model runs more slowly than its untimed model, however, it is much faster than the HDL simulation in development and running time. The model helped the SpiNNaker team take a few more design decisions based on its behavioural results, such as a change in the router design to disable or delay emergency routing to

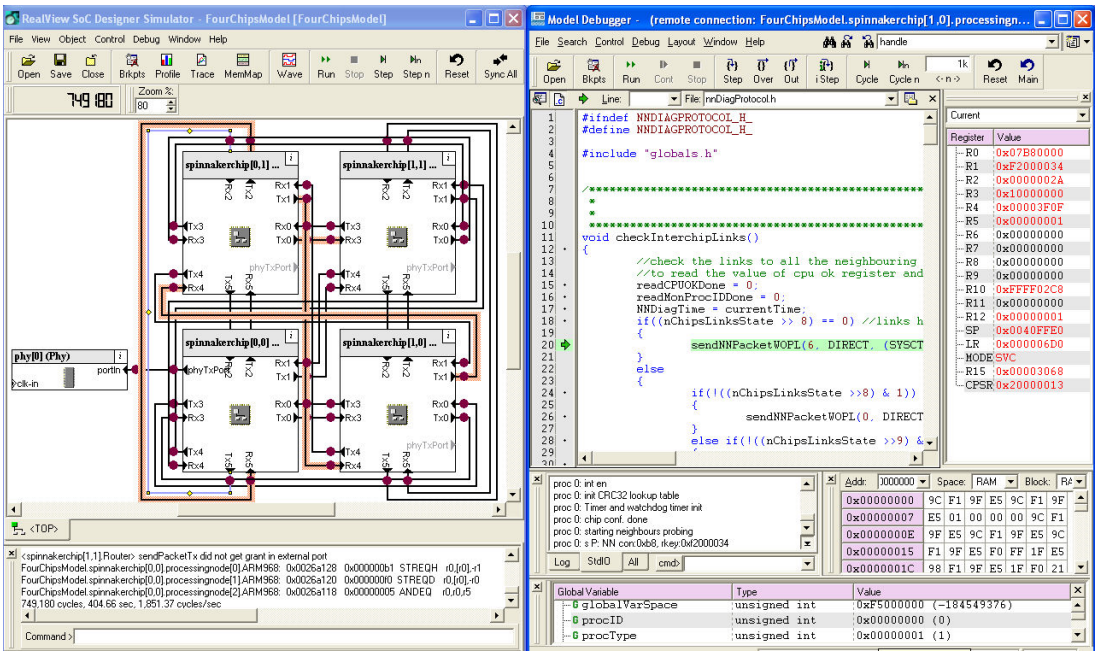

Figure 5.5: SpiNNaker System-level Model (TF-AV(CA)) with Real-time Code and System Debugging.

support Multi-layer Perceptron (MLP) neural network modelling. Besides this, the System Controller's composition was specified based on the envisaged chip-level management functions, the Communication Controller design was improved, the neighbouring chips' communication mechanism using NN packets was clearly defined, and the Ethernet controller's functionality was defined.

5.6.4 SpiNNaker TF-AV(CA) Model

The SpiNNaker TF-AV Model can simulate SpiNNaker inter-chip communication in a cycle-approximate time and supports high-level applications with its high-level processor models. However, in the absence of an ISS model for the ARM processing cores, a real SpiNNaker application could not be developed and run on the model. This was due to the non-availability of a stand-alone SystemC IP for the ARM968 processing core. The TLM model for the ARM968 processor and other ARM-based components required us to run the model as part of the ARM SoC Designer and its simulator. In the third stage of the SpiNNaker system-level model development, we ported all our modelled components into the

SoC Designer and used the ARM-provided SystemC models for all the components provided by ARM such as the ARM968 ISS, Interrupt Controller, Timer, AHB, Watchdog Timer, memories and external memory controller. As the SoC Designer simulates timing at the minimum granularity of a component's clock cycle, without any support for the asynchronous components, we simulated the asynchronous communication in terms of a relevant master component's clock cycles e.g. the 13-15 nanosecond real-time delay at the TX interface has been simulated by 3 router clock cycles at 200MHz clock speed (5 nanoseconds per cycle).

The model can be used for application development, testing and running just like the actual hardware. We use the ARMCC compiler and ARM-ASM assembler from ARM to produce an ARM968 instruction set binary image to be loaded into all the processing cores in the model. We can simulate a single- or multi-chip configuration with varying numbers of processing cores controlled by model-parameters. The model is being used extensively for design validation and application development/debugging purposes by the SpiNNaker team. The model supports debugging for hardware, software or both at once in the real application time (Figure 5.5). We can place breakpoints at the communication lines, buses, memory locations etc for hardware debugging. The code, at the same time, can be debugged with the help of the ARM RealView Debugger at cycle- or instruction-level granularity. The simulation runs in the SoC Simulator GUI, which helps debugging and profiling in an interactive way.

We carried out yet another case study with the help of this ISS system-level model to verify our application developed with the ARMulator [JFW08]. Simulation performance running sample configuration code on the SpiNNaker TF-AV(CA) model vs. the Verilog-based top-level logical model of the SpiNNaker-CMP is shown in Figure 5.6. The results show that the SpiNNaker SystemC system-level model is about 1000 times faster than the Verilog top-level behavioural model for SpiNNaker.

5.7 Functional Validation

Before using the system-level model to validate that SpiNNaker's design meets the envisaged objectives, the model itself was verified for correctness. Extensive functional testing of each component's SystemC model was undertaken comparing

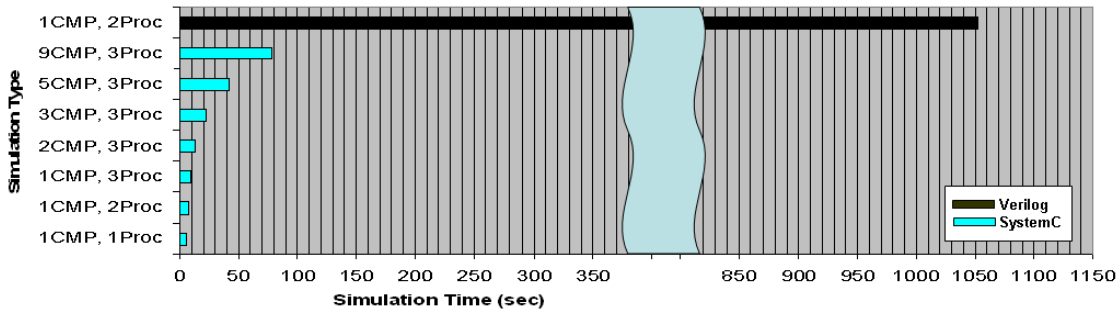

Figure 5.6: Simulation Performance, SystemC vs. Verilog Model.

its results with the Verilog simulation for functional correctness and cycle-level accuracy. Having established a cycle-accurate behaviour for an individual component, it was incorporated into the system-level model. To test the functional correctness of the system at various levels of abstraction we carried out various case studies which gave us a confidence in the system-level model, and also in the SpiNNaker design. These case studies are described in the following sub-sections.

5.7.1 Case Study I

This was a very simple case study designed to verify the communication infrastructure of the multi-chip SpiNNaker computing system with the help of the SpiNNaker UTF-AV model [KLP+08]. In the absence of a spiking neuron application for SpiNNaker, we created a sample application with a varying number of neurons to send spikes to each other with random inter-neuron connectivity. SpiNNaker requires mapping the underlying neural network into its processing nodes. Assignments of neurons to processors can be arbitrary, i.e. any neuron can be allocated to any processor. The routing tables are configured accordingly to establish the defined connectivity among the neurons. The following mapping technique was adapted for this case study.

- As a first step in the mapping process every neuron was assigned to a particular processor in a 16-CMP (each CMP containing 4 application processors) system. Given the absolute flexibility in assignment, this was a heuristically-driven step. Figure 5.7 shows a small (8x8 neurons, *each neuron shown as a black circle*) section of a 2D neural network. In the nervous system, the neural networks exhibit locality of connection, *i.e.*, neurons tend to make most of their connections with nearby neurons, forming clusters;

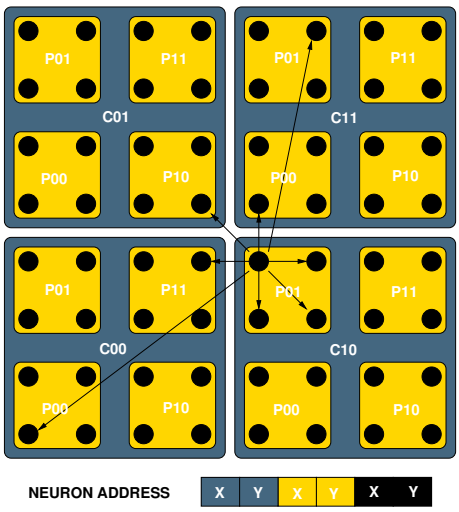

Figure 5.7: Neuron mapping to the processors in a 4-CMP (C00-C11) SpiNNaker system, each CMP containing 4-application (fascicle) processors (P00-P11). A 32-bit neuron's address (shown in the bottom) is formed by placing the CMP ID (X=0, Y=0 for chip C00) in the 16 most significant bits, processor ID (X=0, Y=0 for processor P00) in the next 6 bits, while the neuron ID (X=n, Y=m for neuron Nnm) is in the 10 least significant bits [KLP+08].

these clusters are sometimes known as fascicles. Therefore, assigning neighbouring neurons to the same processor, or processors in the same chip, will result in shorter routes and, thus, less inter-chip traffic. Figure 5.7 shows a very simple, 'rectangular' assignment of neurons to processors (*labelled Pnn*) and chips (*labelled Cnn*).

- The second step in the process was to map each neuron into the SpiNNaker virtual address space. As explained in Chapter 4, the routers can associate neurons in groups and a spike to that group is routed using a single routing table entry in the MCRouter. We identify each neuron by combining the chip ID (chip-X and chip-Y i.e the chip-address's x- and y-coordinate respectively in a 2D chip matrix), the processor ID (proc-X and proc-Y in a 2D processor matrix in each CMP), and the neuron identifier (neuron-X and neuron-Y in a 2D matrix of neurons in a group of neurons being simulated in a processor) as shown at the bottom of Figure 5.7. This mapping of neurons guarantees that neurons which have been assigned to the same processor can be grouped in the same routing entry. By giving proper values to the router entry masks, neurons assigned to different processors in

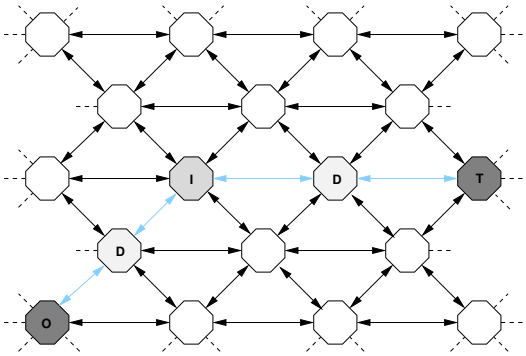

Figure 5.8: Route Setup [KLP+08].

the same chip can also be grouped in the same entry, if adequate.

- Once the neurons have been assigned to processors and mapped into the virtual address space, routing information can be generated. The first step is to set up a route for every connection in the neural net. The process is driven by two criteria: the route should go through the minimum number of routers, and should result in the minimum number of router entries. These criteria should optimize the use of router resources and should also have a positive impact on network traffic. The 'default' routing mechanism, introduced in Chapter 4, was used to eliminate the need for routing entries in most routers along the selected routes. An example which shows how routes are 'setup' is illustrated in Figure 5.8. A packet sent from the 'origin' node (labelled O in the figure) to the 'target' node (labelled T) traverses the predefined route shown. The route is one of the (possibly many) shortest routes available, which will comprise, at most, two straight segments that meet at the 'inflection' or turning node (labelled I). The segments may contain intermediate nodes (labelled D). If a multicast packet is sent along the route described above, routing entries are needed in nodes O, I and T, while the rest of the nodes, *i.e.* D nodes, can take advantage of the default routing mechanism. Default routing can reduce the size of the routing tables significantly, especially in long-distance neuron connections.

- Once a route has been set up for every connection associated with a neuron, the next step combines the individual routing entries into multicast routing entries. This must be done carefully, given that some of the individual routes rely on default routing. As a result of this step, 'primitive routing

tables' are generated, so called because each table has, at most, one routing entry per neuron. In most cases, due to the locality of connections and the use of default routing, routers would not require a routing entry for a large majority of the neurons.

- The final step in the process is to generate minimal routing tables. For this purpose, the tables were treated as logic functions: the multicast entries constitute the on-set of the function, the default entries constitute the off-set and 'unrelated' neurons, *i.e.*, those that have no routes traversing the node, were considered the 'don't-care' set. An application called Espresso [RSV87], a well-established logic minimizer, was used to minimize the tables.

An application called 'SpiNNit' was developed to automate the process and this was used in the generation of the expected results to be compared for validation. For this case study, we mapped 4 neurons to each fascicle processor. Each chip contained four fascicle processors while the system consisted of 4x4 (16) chips. The neurons' mapping information was provided to SpiNNit to determine the routing table entries for the on-chip routers' multicast lookup and mask tables. Sample packet files for each processor were created by SpiNNit with expected output files containing the packets each fascicle processor should receive at the end of the simulation. The simulation recorded the output packets with one output file for each processor. While the fascicle output files recorded the received packets, monitor output files contained any packet dropped due to some error such as long lasting congestion, parity error, or time-phase error. It was difficult to check all output files manually against their expected output files and then trace any unexpected or dropped packets. An automated process was created with the help of the transaction recording functionality of the SystemC Verification (SCV) library. This compared the output with the expected outputs and generated a report for any missing or unexpected packets found in any fascicle output file. The experiment was run using the SpiNNaker system-level model several times with many variations for debugging and verification purposes and, after some debugging and bug-fixing, the results matched the expected behaviour.

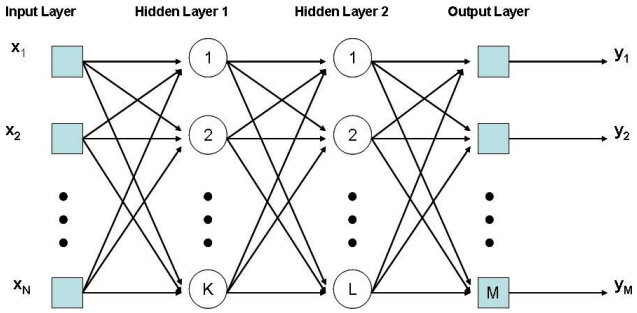

Figure 5.9: A Typical Multilayer NN Model [KLP+08].

5.8 Case Study II

This case study was performed as a feasibility study to test a rate-coded Multi-Layer Perceptron (MLP) neural network application on SpiNNaker [KLP+08, KJFP07]. The application was run on the SpiNNaker TF-AV(CX) model for the communication results and ARMulator for processing results. A typical MLP neural network learning algorithm consists of an input layer, an output layer and a number of hidden layers each containing neurons with the connectivity as shown in Figure 5.9. Psychologists use a similar model, known as a Parallel Distributed Processing (PDP) for neural network learning in the psychology world. We carried out this case study in collaboration with researchers in the School of Psychological Sciences, at the University of Manchester. They have been using a PC Cluster to run the simulation with an application called the Light Efficient Network Simulator (LENS) [Roh99]. In MLP models, every neuron in a layer can receive input from many neurons (depending on the connectivity level) in the preceding layer. The inputs are multiplied by the connection weights and then summed to produce the output to the next layer. The output layer neurons add the weighted inputs to generate the activation using an activation function. The output is used with the target output to compute the error (delta) to back-propagate synaptic updates. The PDP model is quite different from a spiking neural model, as the packets have to carry a payload as input to the next layer and there is no notion of real-time processing involved. Moreover, many inputs can converge at the same target neuron at the same time, which may cause a bottleneck for the Communication Network.

These variations complicated the task for the communication system of SpiNNaker. However, we used a different mapping scheme and the flexibility in the

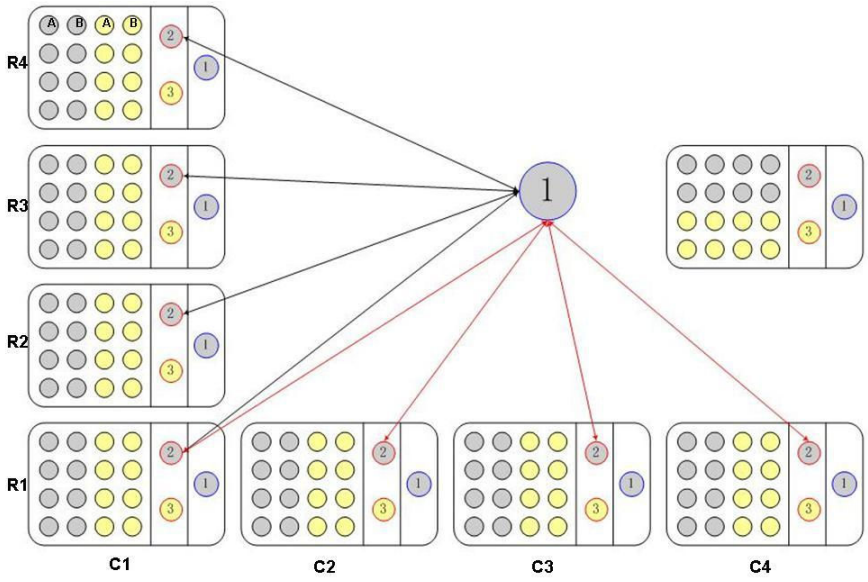

Figure 5.10: PDP Neural Mapping [KLP+08].

SpiNNaker architecture to circumvent this problem. Instead of using a direct, one-unit-to-one-processor mapping with the inherent risk of traffic congestion at target units, we cast the problem as a matrix multiplication problem. Consider the neurons as partial result-computing units and feed the results forward to the next layer units which accumulate these results and then pass them to those further ahead, until they are received by the output layer to compute the activation and delta. The same process is repeated in reverse for the back-propagation of the delta.

We divided the large matrix computational problem into small portions, assigning one portion to each processor out of the many spread over different chips. To further reduce the router traffic we decided to pass on results among the processors on the same chip using a shared-memory message-passing technique with the help of on-chip System RAM. The partial results sent to the processors outside the chip were passed as multicast packets after configuring the routing tables as per the procedure described above. As shown in Figure 5.10, the dark coloured and light coloured processors in processor columns marked as 'A' in each chip compute partial products from the input vector and the weight matrix, and pass these to their corresponding coloured processors marked as number 2 and 3 respectively using a shared-memory message-passing technique. Processors number 2 and number 3 on each chip in the chip columns 'C1' to 'C4' send their

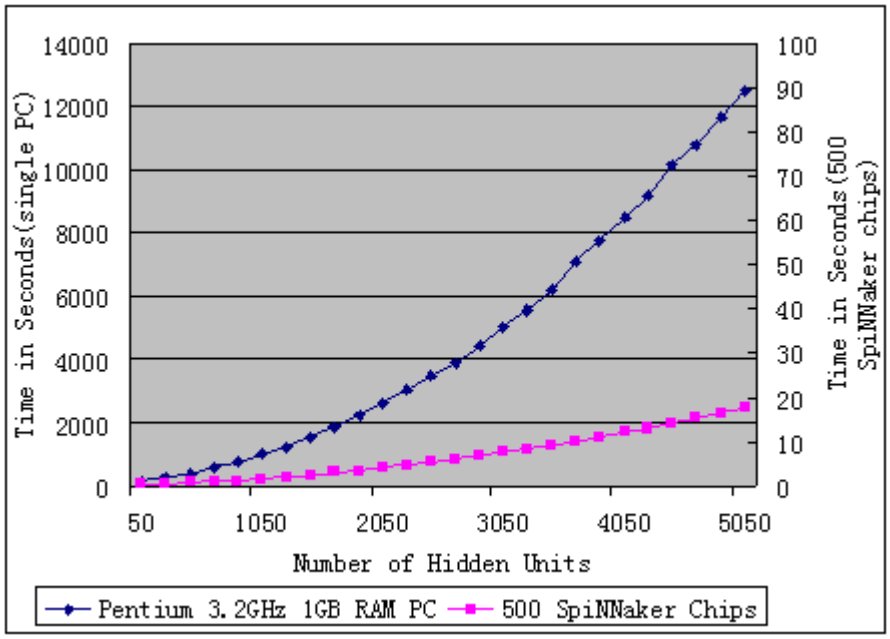

Figure 5.11: Simulation of the PDP Model on the SpiNNaker System [KLP+08].

results to processors number 1 in rows 'R1' and 'R2' respectively using multicast packets. After a specified interval, the same procedure is repeated for dark and light coloured processors in processor columns 'B' on each chip. The processors number 1 are the output layer processors that compute the activation output and the partial delta values. In the second phase of the feed-forward pass, processor 1 of each chip in chip-column 'C1' will multicast the partial delta result to all number 2 processors in row 'R1', processor 1 in column 'C2' will multicast to all number 2 processors in row 'R2' and so on. During back-propagation, the computed delta is transmitted backward following the reverse path.

The routing tables for forward and backward path multicast packets were computed with the help of SpiNNit for various scales of the SpiNNaker system. Because the ARM968 core was modelled at an abstract level in the SpiNNaker TF-AV(CX) system-level model, the computation was performed using the cycle-accurate ISS model of ARM968 processor in the ARM Ltd. RealView ARMulator to acquire accurate computational timing including the time for shared-memory message passing. We implemented the PDP model in C++ for the system-level model incorporating the processing delays acquired from RealView ARMulator and simulated a 195-chip SpiNNaker computing system. As the model gives

Table 5.1: Performance of PDP on PC vs. on SpiNNaker [KJFP07].

Exp. no.	Hidden units	3.2 GHz PC	500 chips SpiNNaker	Speed-up
1	50	151 (s)	.42 (s)	357.86 (times)
2	450	394	.77	509.44
3	1050	993	1.63	609.28
4	2050	2613	4.04	647.23
5	3050	5040	7.51	671.28
6	4050	8497	12.04	705.61
7	4450	10126	14.15	715.42

accurate timing for the communication, the resulting simulation time was approximately what we expect on the SpiNNaker hardware. The simulation results with a comparison to the LENS simulation on a PC are shown in Table 5.1 and Figure 5.11. In this case study, as before, we managed to keep the routing table entries to a minimum with the help of default routing, masking and shared-memory message-passing techniques.

5.8.1 Case Study III

We carried out this case study with the help of the SpiNNaker TF-AV(CA) model using the ISS model for the ARM968 processing cores. The purpose of this case study was to verify the design objectives of the SpiNNaker computing system by running a spiking neural network application on a simulated multi-CMP SpiNNaker machine. The application was developed with the help of the ARM ARMulator ISS model for a single ARM968 processor, and then tested on the SpiNNaker multi-CMP system-level model. It is an event-driven real-time application [JFW08] using the Izhikevich spiking neural dynamics model [Izh03a]. The application simulates 1000 neurons on each processing core with a random neural connectivity defined by the routing tables. The synaptic weights and axonal delays for inter-neuron synapses are chosen randomly and stored in the SDRAM with each chip. The spikes are passed among the neurons as multicast packets which are routed by the on-chip routers to relevant on-chip/off-chip processors. The application is based on the SpiNNaker standard application model explained in Section 4.5 of Chapter 4. The same code developed earlier for the ARMulator was run on a single-CMP and a 4-CMP SpiNNaker system-level model, each

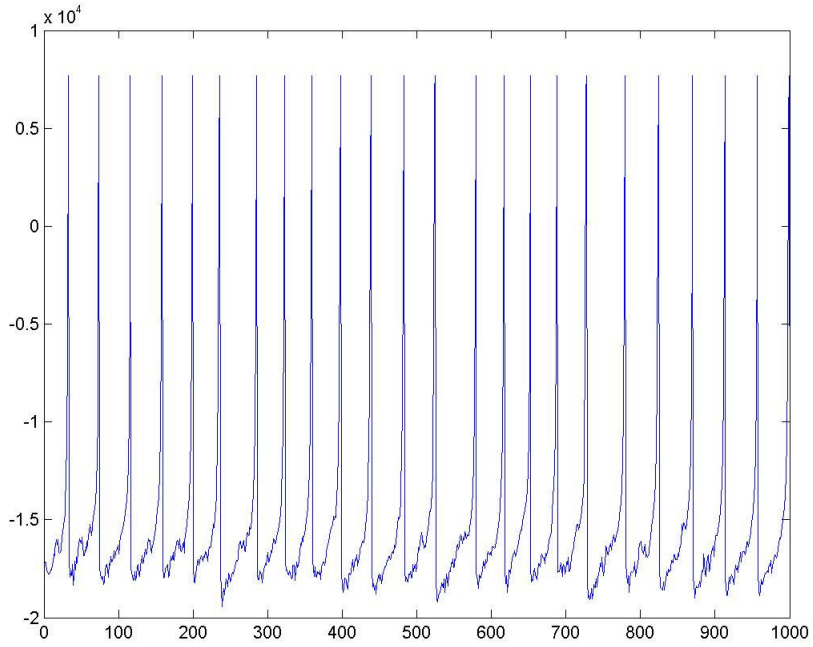

Figure 5.12: Spike Train from Izhikevich Neurons [JFW08].

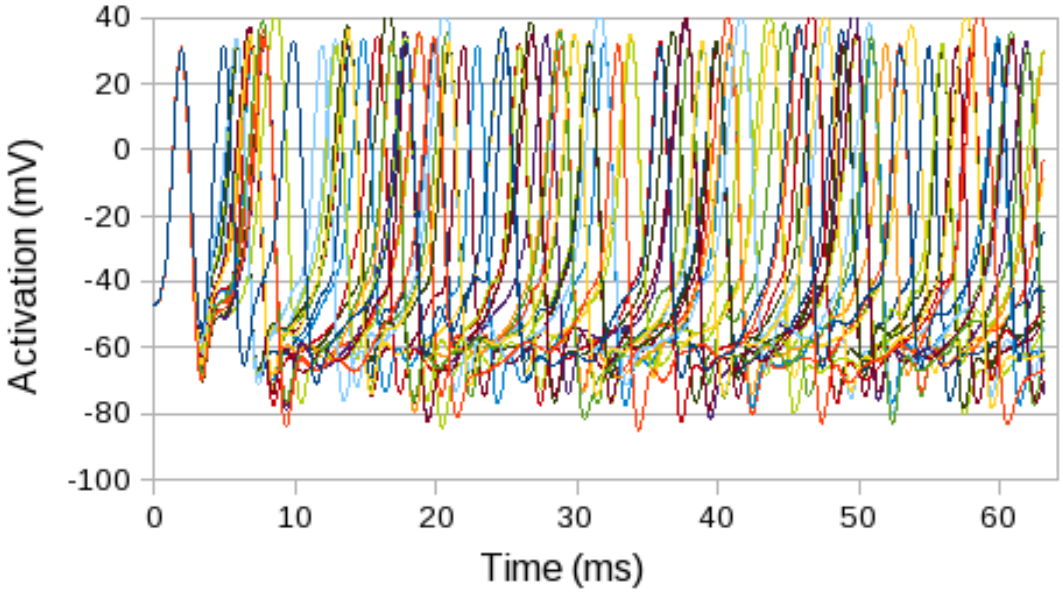

Figure 5.13: Spike Train from Izhikevich Neurons with a 4-CMP SpiNNaker System-level Model [RKJ+09].

CMP containing three processing nodes. The neural mapping was worked out with the help of an application which generated the synaptic-data look-up tables for data TCM and routing tables entries for on-chip routers according to the defined mapping. The application along with the routing tables was loaded into the CMPs and run for varying intervals of time. The neurons' initial state was chosen randomly while keeping a few neurons in a hyper-active state to stimulate initial spike activity in the system. Each neuron's spike activity was recorded in the output files for spike pattern analysis. The results for a single chip simulation are shown in Figure 5.12 and those from a 4-CMP simulation are shown in Figure 5.13. Results are comparable with those produced with the ARMulator for a single processing core [JFW08]. These results provide satisfactory verification of the design and functionality of the SpiNNaker system.

5.9 Hardware Functional Verification

The SpiNNaker system-level model has been used extensively in the design verification of the SpiNNaker CMP. A top-level RTL model for the test chip has been created comprising two processing nodes and all the chip-level components for exhaustive component-level and integrated testing before sending the chip for fabrication. The system-level model has been used as a golden model for the functional verification of the hardware components. The test cases were prepared and tested on the system-level model before executing on the RTL top-level model. The process helped spot a few critical problems in some components which could not be spotted earlier during component-level RTL validation. Similarly, a few bugs traced in the system-level model were also removed to match the functionality of the two models. Detailed testing of the Communication Network including the MCRouter and the Communication Controller, the DMA controller, and the Ethernet Controller was carried out during this process. Besides testing these components, the tests helped in understanding and refining the behaviour of these subsystems as part of the SpiNNaker CMP.

The model is also being used for application development for the SpiNNaker computing system. Ongoing work on the SpiNNaker standard application using the Izhikevich model, the implementation of Spike Timing Dependent Plasticity (STDP) on SpiNNaker, and refinement of the Parallel Distributed Processing

(PDP) model are using the SpiNNaker system-level model for application development and testing.

5.10 Summary

A complete system-level model gives a holistic view of a computing system's functionality while the hardware is still in its design phase, which is not possible with component-level simulation. Simulating a large-scale massively-parallel computing system of the scale of SpiNNaker with conventional HDL is very difficult in terms of performance and development time. Presently, commercially-available complete system simulations do not suit system-level modelling for SpiNNaker due to their limited scope or proprietary nature. For these reasons a complete system-level simulation of the SpiNNaker massively-parallel computing system was developed using the SystemC TLM technique as part of the SpiNNaker research project. The model contributed effectively in refining the architectural design, evolving/resolving system-level functionality issues, and verifying the functionality of the system at component and system level. The TLM design principles have been adopted while evolving the system-level model from algorithmic modelling (UTF-PV) to the cycle accurate instruction set simulator with full architectural functionality (TF-AV(CA)). The end product is very close in functional behaviour to the actual hardware. We have been able to develop, debug and test applications intended for the actual SpiNNaker computing system, while the system is still in the design phase. Extensive testing has been carried out to verify that the SpiNNaker system-level model exhibits a behaviour close to that of the actual hardware. The system-level model helped verify the design objectives of the SpiNNaker computing system through the use of case studies to test two quite different potential applications for SpiNNaker.

Chapter 6

Multi-CMP Systems Configuration

6.1 Introduction

Over the last few years, miniaturization in computer manufacturing has shrunk the size of computing devices to fit in a pocket. This is a direct consequence of high-level integration motivated by customer demand [Fur05]. On the basis of these industrial trends, we can predict the future of computing devices in the years to come. A direct implication of this development is the integration of many independent components in a system to form a System-on-Chip (SoC); a concept that has moved the manufacturing industry forward beyond conventional Very-Large-Scale-Integration (VLSI). Technology today allows the incorporation of hundreds of intellectual property (IP) blocks onto a single high-performance SoC. High-performance computing machines designed to run complex applications, such as databases, web or search engine servers, mathematical modelling, and scientific applications etc, have started to use the same technology. Instead of using a single powerful and complex processor with floating-point and multimedia functionality, SoC designs are adapting to use multiple simple processors to run these applications with process- or thread-level parallelism.

The Chip-Multiprocessor (CMP) design is a combination of the two approaches i.e. SoC design to integrate multiple resources on a single chip, and the grouping of multiple simple processing cores onto a chip for multiprocess/multithreaded

computing. Besides avoiding the micro-electronics bottlenecks in designing complex high-end processors with very large number of transistors, this is a consequence of the requirements of present day commercial application which are mostly transaction-based and can be run in parallel [BKM+00]. The CMP adopts a hierarchically-partitioned design with replicated modules on a chip, allowing the use of short interconnects to improve performance [BKM+00, HNO97].

CMP-based design brings with it some design- and application-level challenges, such as design complexity, on-chip interconnections, area management, power and heat management, inter-process synchronization, process/thread mapping to the CMP processors, debugging support, and configuration for optimal use of resources to support parallel distributed applications [CFBC06, CACY+06, LM06, LBHS06, LLB+06]. The issues are more challenging in a multi-CMP computing system; a completely distributed computing system mostly without central shared resources and synchronisation mechanism.

This chapter highlights some multi-CMP system configuration challenges and presents the configuration process in two well-known large-scale multi-CMP computing systems to examine the approaches adopted to deal with these issues. To conclude, some peculiarities of SpiNNaker, a truly multi-CMP system, are given to justify the reasons for devising a novel configuration process for SpiNNaker.

6.2 CMP Configuration Challenges

Multi-CMP computing systems are characterized by their concurrent processing and inter-process communication mechanism to run parallel distributed applications. From the application's perspective, the most important issue in a CMP-based system is the inter-process synchronization, and to use a large multi-CMP system as a unified computing system for running a single large application [MBH+05]. In CMP-based systems, processing nodes are independent functional units centred around a processing core with dedicated supporting peripherals to support parallel distributed computing. There is no central system-wide clock as the CMPs are connected via an asynchronous network to form a multi-CMP computing system. A multi-CMP system can be viewed as being similar to a PC-cluster running a single or multiple applications in parallel. We need to configure these systems to have a unified system-wide view and to support the running of large applications in a distributed manner. The configuration process

in such a system involves initializing each CMP's components, configuring the intra- and inter-CMP communication infrastructure, configuring all the CMPs to work in collaboration, configuring the system to interact with the outside world, loading an application (or a number of applications with system-level partitioning), and providing a means for the user to interact with the application at run-time. Besides these, the process needs to provide component- and system-level fault handling, as a large-scale multi-CMP system may suffer run-time faults. The following are some of the major issues involved in the configuration of a multi-CMP computing system.

- System-level Synchronization: In a system comprising multiple CMPs, a large number of concurrent processors are working autonomously without central control. For a single parallel distributed application, however, the processes need to synchronize their activities to maintain a unified application state. For example, in a cluster of PCs, individual processes may compute chunks of result and pass these to some other processes which accumulate these results, which may then be passed to a process producing the final result. There is a need to synchronise this activity as the outputs from some processes may depend on inputs from others. Synchronization is achieved using inter-process communication. To run a real-time spiking neural network on a multi-CMP system, for example, it is important to synchronize the processes at millisecond temporal granularity to model biological behaviour. Similarly, in the SpiNNaker Communication Network, if we want to discard free-wandering errant packets based on their timephase, we need to have a synchronized notion of time (within acceptable time bounds) across the whole system.

- Inter-process Communication: In a multi-CMP system shared-memory message-passing communication is not possible between the processors in different CMPs in the absence of a shared global memory. The only way processes can communicate across the system is by passing messages as packets. For packet-based communication, the communication infrastructure needs to be configured and the processes need to follow a communication protocol as per the network configuration. The configuration process also needs to support the application in sending/receiving and interpreting these packets. Besides supporting the application, the communication infrastructure

is also important for supporting the user's interaction with the application and the system.

- Chip-level Initialization: During power-on self-test the chip components must be tested and initialized by the processors. This requires a chip-level coherence among the concurrently-running on-chip processors. This may be achieved by hard-wiring a bootstrap processor in each chip at design-time to configure the chip-level shared resources, instead of all the processors trying to do the same job. Similarly, the chip can have a Boot ROM to contain initial bootstrap instructions to test and initialize the processing nodes and the CMP resources. However, reliance on a single boot-processor and on a Boot ROM to initialize a CMP may introduce a single-point-of-failure which may render an otherwise healthy chip non-functional. The chips should also be initialized to interact with other chips and the outside world.

- Application Loading: A multi-CMP computing system should be able to support the running of a variety of applications and be scalable to match computational requirements. As there is no centralized storage and each processing node needs to use its dedicated application image to run a part of the application, the configuration process should be able to load the application dynamically into the system. We also need to devise some way to partition the system to run multiple applications simultaneously or to use the whole system for running a single large application. Additionally, the application needs to be loaded to individual CMP memories so that the processing nodes can run the application with the relevant data readily available. This can be done with the help of the system's connection to the outside world and the communication infrastructure available, or a separate dedicated network is required in addition to the application-specific inter-CMP network, to load the application and interact with the application from outside the system.

- Fault-handling: A large system comprising numerous processing elements and memory resources may get some expected or unexpected faults at run-time. It is difficult to trace a fault in a large-scale parallel distributed system and rectify it at run-time without disturbing the system's state. A centralized error recovery mechanism may itself introduce errors. The best solution is to provide distributed autonomous fault-detection and recovery

mechanisms at various levels to detect the faults and to recover the system at run-time, or isolate an irreparable component to avoid disturbance to the rest of the system. This requires the configuration process to provide an autonomous chip- and system-level fault-handling mechanism and support for reporting unhandled faults to the the user.

- Connection to the Outside World: A CMP-based system needs to communicate with the outside world for application loading, user-level interaction with the application, and visualizing the state of the application. Various techniques with inherent pros and cons can be used for this purpose such as connecting each CMP to a central service node with the help of a dedicated network, or connecting a service node via the already-available network connecting all the CMPs, etc.

- Chip-/System-level Monitoring: To manage the system and the application, we need some system/application monitoring and management facility for the user. Either a part of the management application can run on each processor to support the monitoring, or a processor on each chip be dedicated to perform as a "monitor processor". The later is a better option as it allows the rest of the processors on each chip to be dedicated to the application execution without run-time disruption. The monitor processors in a multi-CMP system can be used for system-wide management in a distributed manner. A distributed operating system (OS) can also be run with the help of these processors across the whole system for better system-level management. The monitor processor can also be used for CMP-level coherence and chip- or system-level fault handling, and reconfiguration.

- Interactive Control: The user needs to interact with the application/system at run-time for exchange of data, to debug an application, diagnose the system for performance or faults, or to view the state of the application at some point in time. It is very difficult to achieve a system-wide coherent state of the application at run-time in a large-scale dynamic massively-parallel computing system.

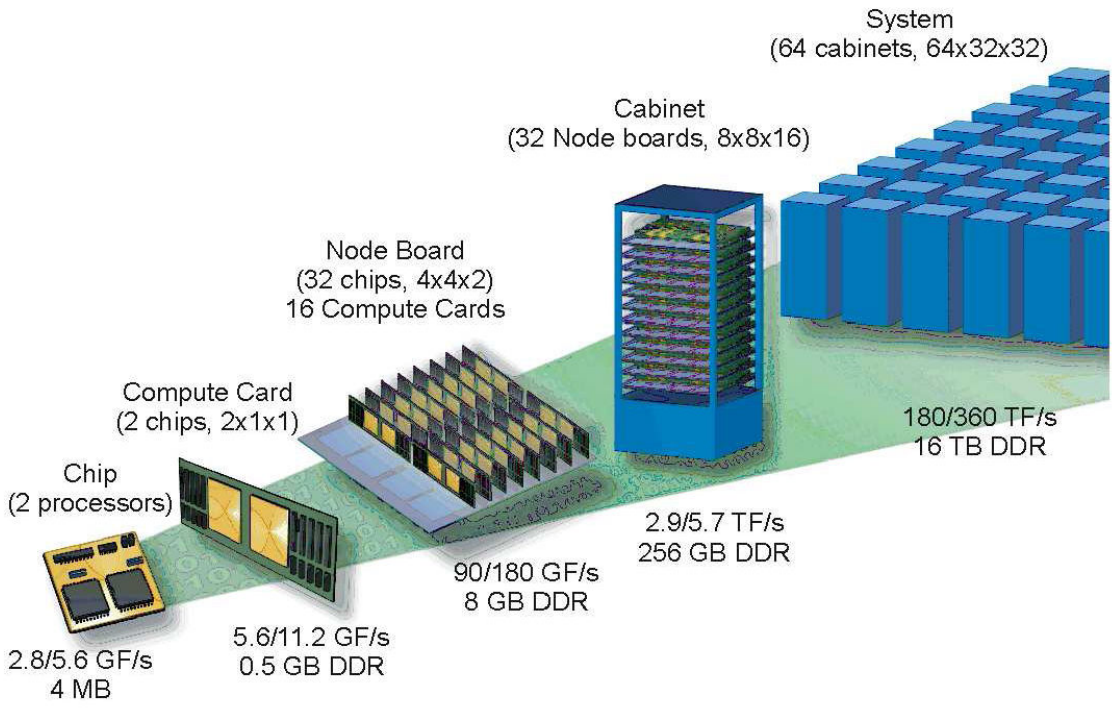

Figure 6.1: Blue Gene/L - System Overview [ea03].

6.3 Related Work

The following sections present the configuration process in two large-scale massively-parallel multi-CMP systems to examine how these architectures handle the configuration issues to support large-scale parallel distributed applications. The two architectures have been selected as being large-scale multi-CMP system interconnected in a torus by connecting each chip to six neighbouring chips using a packet-based network. Similar to SpiNNaker, the two architectures have been designed to run parallel applications distributed on their independently-functional compute nodes with dedicated memory and other resources.

6.3.1 Blue Gene Configuration Process

IBM's Blue Gene/L (BG/L) supercomputer comprises up to 65,536 CMP nodes designed around embedded PowerPC processors [HBB+05]. The system gives an overall peak computing power of 180 or 360 teraflops depending on the utilization mode. The BG/L represented a new level of scalability for parallel systems with the largest integration of computing nodes of its time; it was rightly declared the

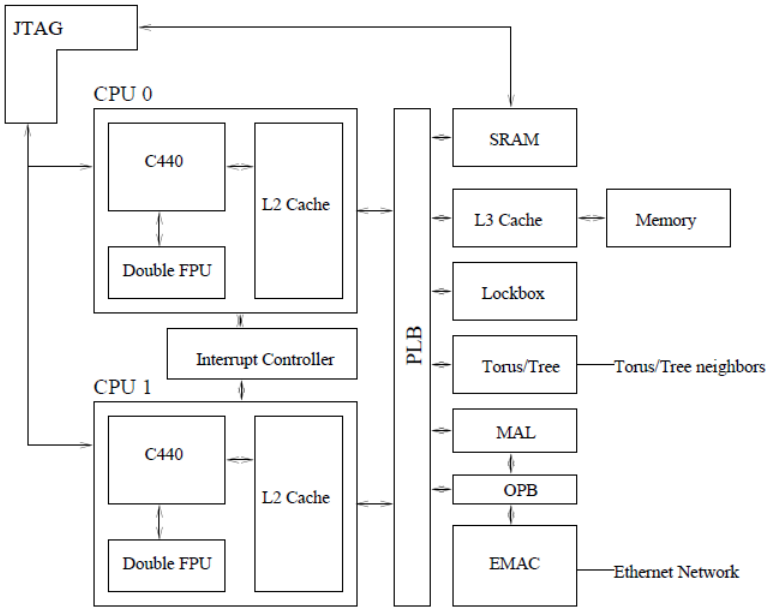

Figure 6.2: Blue Gene/L - Compute Node's Block Diagram [ea03].

fastest supercomputer in the world until 2007.

Functionally, the BG/L system can be divided into three parts: the Computational Core, the Control Infrastructure and the Service Infrastructure [GBea05]. The Computational Core comprises 64Ki[1] nodes, each consisting of 2 PowerPc400s running at 700 MHz. One of these is used for user applications while the other is used for configuration, networking support and chip management. Two such compute nodes share one "node card" with additional SDRAM. Each compute node has six bi-directional links for connection with other compute nodes and these are interconnected using a 64x32x32 three dimensional torus [ABCea05]. The network provides reliable, deadlock free, but unordered delivery of packets (up to 256 bytes) using minimal adaptive routing and broadcasting. The system can be partitioned into several independent subsystems capable of performing independent jobs. The Control Infrastructure comprises 1024 I/O nodes which are like compute nodes but with more memory. This infrastructure is used to provide an interface to the computational core to interact with the outside world. Each I/O node is a Linux box to facilitate user interaction with the system using an Ethernet connection [OBea05].

To a user, the system appears as a 1024 node Linux cluster, each node a 64-way

[1] 1Ki=1024

multiprocessor machine. The Service Infrastructure comprises varying numbers of service nodes to form a traditional Linux cluster which resides outside the BG/L Computational Core. The service nodes communicate with the computational core with the help of I/O nodes using packets over a separate dedicated Ethernet. This infrastructure is used for configuring, controlling and monitoring the system. I/O nodes use Linux while the compute nodes use the BG/L Run-Time Supervisor (BLRTS) kernel. These Operating Systems (OS) are local to the nodes while the global OS runs on the service nodes and is responsible for booting up, monitoring and job launching. A Database-Management-System (DBMS) with a central database is used over the global OS to maintain the state of the system and system partition information which is updated periodically [HBB+05].

There is no Boot ROM in the compute nodes, rather the whole initialization is done at run-time by the service nodes through the Ethernet [HBB+05]. A small boot loader is first written directly to each nodes local memory by the service nodes. This loader receives and loads the boot image into the memory using packet-based messages from the service nodes. The boot image is the same for all compute nodes while the I/O nodes have a second type of boot image. The size of this common boot image broadcast to these nodes is 64 KB for the compute nodes and 2MB for the I/O nodes. After this common information, each node is "personalized" by assigning distinct torus coordinates, a tree address and an IP address. The BLRTS implements a system call to request the node personality. System monitoring and job execution is done by a combined action of the service and the I/O nodes which maintain log files to this effect. For a particular job the user specifies the size and shape of a partition and based on this the global OS selects a set of compute nodes to form the partition.

The booting-up process in Blue Gene/L is based on the concept that the service node can control the computational core to the lowest level of granularity [HBB+05]. The communication between service and compute nodes is based on packets using Ethernet, which are dropped at a control FPGA chip attached to each node card. The control FPGA converts the packet based communications to JTAG (IEEE 1149.1) commands for the compute nodes. One control FPGA can drive up to 36 node cards. The packets containing a common boot loader and boot image are broadcast through this infrastructure. Similarly information collected from both types of nodes such as faults or acknowledgments etc. is transferred back to the service node by these control FPGA chips in the form of

User Datagram Protocol (UDP) packets. For this purpose each compute node provides an extension to the basic design in terms of JTAG compliant ports and supporting hardware [HBB+05].

Besides this, certain system-level configurations are also performed such as control register initialization and Test Access Port (TAP) controller configuration to support initial boot-loader loading into the memory and non-intrusive access to device control registers for control, debug and monitoring purposes. The JTAG on each PPC440 can directly access a compute nodes 16Kbyte portion of L2 SRAM logically located at the top of the 32-bit decoded address space of the PPC440 and contains the default reset vector. It contains the instructions to be executed after startup or reset. JTAG writes the boot code from the service node at this place for PPC440 boot-up. The same facility of directly accessing this portion of the memory is used by the service nodes to do other control and house-keeping functions. This communication is given the highest priority in the arbitration to make the boot-up and control functions very efficient, however, it does not affect other traffic because of its small packets size. This feature is useful for handling run-time problems and investigating any boot-up issues. After the Ethernet connection is established, the same JTAG communication is used to load the application into the compute nodes' memories.

For fault-tolerance a self-test mechanism is kept in each chip to perform system diagnostics [HBB+05]. During initial built-in self-test, chips failing to pass all tests are marked as dead-chips. The service node also provides a system-debug facility to the designers/system-administrator with the help of a debug I/O port on each compute node and used only when Ethernet is not enabled. Several on-chip signals of interest to the chip logic designer are multiplexed onto the debug port. The operation of the debug does not interfere with the normal operation of the chip.

6.3.2 Cray XT3 Configuration Process

The XT3 is Crays third-generation massively-parallel processing system [ea08] designed to run large-scale parallel applications. This is a multi-node system connected in a 3D torus through Cray XT3 interconnect and the Cray SearStar routing mechanism (Figure 6.3). Its Compute Processing Element (PE) is an SoC based on a dual-core AMD Opteron processor coupled with its memory and dedicated communication resources. Another type of PE, the Service PE, is

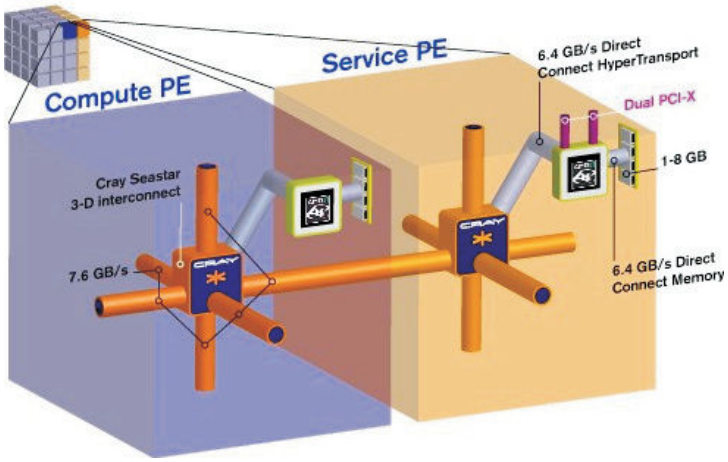

Figure 6.3: Cray XT3 Massively Parallel Computing System [ea08].

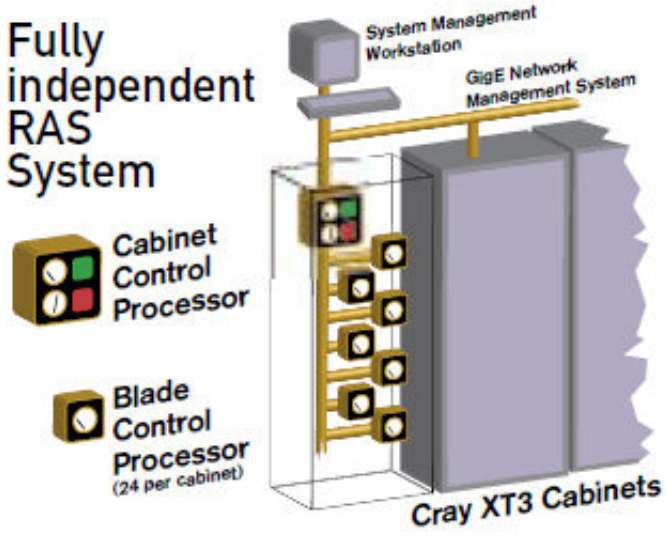

Figure 6.4: Cray XT3 - System Overview [Inc05b].

used for I/O, login, and network/system management functions [Inc05b]. Four Compute PEs are grouped into a compute blade for scalability. Each Compute PE is attached to a SearStar communication chip containing a DMA engine, hyper transport links, a communication and management processor, an interconnect router and service port. An on-chip router on the communication chip connects each node (compute node and communication chip) to six neighbouring nodes in a 3D torus topology with a peak link speed of 7.6 GB/s. The router also contains an error correction and retransmission mechanism [Inc05b].

The DMA engine has its own PPC440 processor for message preparation and demultiplexing tasks to leave the Opteron free to focus on computing tasks. The operating system on Opteron works together with the DMA engine to provide a direct path for an application to communicate to the hardware without any interruption or traps. In case of a bad connection, a link can still be made operational by configuring it to run in a degraded mode. The system is highly scalable and can be made to combine in various numbers of system cabinets. A 320 cabinet system contains 30,508 compute nodes, 106 service nodes, 239 terabytes of memory, and is connected via a 40x32x24 3D torus interconnect (Figure 6.4). Each Compute PE contains 1 to 8 Gbytes of memory with a 1 Mbyte cache to serve the Opteron microprocessor. Each node is connected to the system interconnect at 6.4 Gb/s. The interconnect is devoid of any switches etc which saves cost and complexity while connecting tens of thousands of nodes.

Each communication chip provides a service port which connects each Compute PE to a separate dedicated management network (Ethernet) and bridges between the management network and the system interconnect [Inc05b]. This allows the monitoring system to have access to all registers and memory in the system. The management network is used for boot-up, maintenance and monitoring of the system during initialization and while running applications. A proprietary CrayXT3 Catamount microkernel is designed to run on the compute nodes and provides a computational environment without too much overhead, besides helping the system to scale to tens of thousands nodes. The microkernel interacts with the application in a very limited manner and provides fine-grain synchronization, high performance and low-latency communication. It is used only for scheduling, memory management and virtual memory addressing.

The Service PEs run a full Linux OS and can be configured to provide login, I/O management, or any system/network services. Applications are run using a

login service node on dedicated sets of compute nodes which are partitioned by the system administrator. Fast interconnection with no contention in the memory modules provides a fast boot-up mechanism with minimum downtime [Inc05b]. The Cray configuration/management process integrates hardware and software components to support run-time application execution, system monitoring and fault-handling. An independent monitor system with separate processing and network resources monitors and manages major resources in the Cray XT3 system. The same system is used to manage the interconnect, to monitor and display the system state to the system administrator, and to power up and down the system. The system-level recovery in the case of any malfunction is also managed with the system management system. The management system does not use any resources from the system and recovers any part of the system from failures while the system is still running. Single-points-of-failure have been minimized with redundancy or alternative workarounds in the design and hence the system remains operational even if a (non critical) portion of it is non functional. The communication chips, processors, and memory units etc are all field replaceable and upgradeable [Inc05a, Inc05b, AKB+07a].

6.4 SpiNNaker - A Novel Architecture

As explained in Chapter 4, SpiNNaker has been designed as a massively-parallel multi-CMP system consisting of up to 64Ki SpiNNaker CMPs, each comprising up to 20 independent functional processing nodes. Although the architecture is based on SoC design, it is quite different from other SoC-based multi-chip architectures (Blue Gene/L and Cray XT3). The Blue Gene/L processing node consist of only two processors with one processor already hard-wired for control and boot-strap while the other is used as an application processor. Similarly, in the Cray XT3 the boot process is performed with the help of the service processor on the communication chip which is controlled externally by the service node with the help of the management network. This introduces a single point-of-failure as the chip cannot be brought up to the running state if the service (monitor) processor is dead. In the Blue Gene/L system, there is no Boot ROM, the boot loader and the boot image is loaded at run-time with the help of an Ethernet. However, the process requires the connection of all the processing nodes with the help of a separate Ethernet. This duplicates the networking in the presence of a

64x32x32 torus already connecting the compute nodes at the cost of extra energy and maintenance.

In SpiNNaker, we rely only on the inter-chip interconnect for communication among the CMPs, which reduces the overhead of maintaining large networking resources and keeps the system size manageable. The Blue Gene and Cray XT3 both use Internet Protocol by assigning IP addresses to their computation nodes which introduces computation and communication overhead with extra memory required to run the IP protocol at each chip. In SpiNNaker we use low-power processing cores (200MHz ARM vs. 750MHz PPC440 or 2.6GHz AMD Opteron) with sufficient local memory to run application-specific code efficiently at much reduced power consumption. The application is distributed over all the application processors concurrently and does not require any software scheduler or operating system (Chapter 4 Section 4.5). In both the above mentioned systems, the compute nodes communicate with the outside world with the help of a separate management network, while in SpiNNaker the system is connected to the Host PC with the help of only one (or a few for redundancy) chip(s) from where the same inter-CMP interconnect is used to pass messages between the Host PC and any chip in the system. Unlike BG(L), the SpiNNaker interconnect does not require any switches or network infrastructure to connect its CMPs, rather the chips are interconnected directly through their six I/O links to form a network of CMPs.

These peculiarities of SpiNNaker distinguish it from other CMP-based systems and require a novel configuration process to be devised. The next section describes some of the configuration requirements pertinent to the SpiNNaker multi-CMP system.

6.5 SpiNNaker Configuration Requirements

SpiNNaker can support neural simulations with a variety of neural network types and application models. The system is a generic processor resource: "a universal neural network architecture". There are, as a result, several key system configuration considerations to enable it successfully load and run a given neural network [KNR+09].

- Selecting a "Monitor Processor": For chip-level management, we require one processor out of the 20 on each chip to be the monitor processor. The

monitor processor has 4 roles: pre-boot chip-level configuration and testing, boot-time system-wide configuration, run-time chip-level fault handling, and supporting run-time system-level management. It must perform these management tasks without interfering with the application processors running the neural application. SpiNNaker chips, however, do not have a dedicated monitor processor, therefore, the boot process needs to select a processor per chip to perform this job.

- Breaking System Symmetry: To route a (P2P or MC) packet to its destination, the chips need unique addresses. SpiNNaker chips are identical with no hard-wired addresses, while the SpiNNaker system is organized as a symmetric toroid with no starting point. Before any application can run on the SpiNNaker system, we need to configure the chips with unique addresses to enable inter-process communication.

- Run-time Configuration: To conserve the CMP area, the Boot ROM size has been kept to a minimum, just sufficient to support initial testing and device initialization. The remaining CMP- and system-level configuration must be performed from outside the system for better flexibility and fault-tolerance. We need to configure the CMPs to collaborate with each other to form SpiNNaker as an integrated system. It is, however, desired that the Boot ROM should not introduce a single point-of-failure.

- Application Loading: SpiNNaker CMPs are not pre-configured to simulate a particular neural dynamic model; they can simulate arbitrary spiking neural models. This means the user initially configures the application outside the system, then loads it. Hence, we need a detailed methodology to load the neural application with associated data to each chip in an efficient and scalable way.

- Network Configuration: The configuration process needs to configure the on-chip router as per the neural mapping (i.e. the neurons mapped on the processors by the application) to conform to the neural network being simulated. A neural network is simply a configuration, not in itself a running application. Much like an FPGA, the boot process can configure the network once and then the target application can be run many times without reloading the system. Therefore although a fast boot time is desirable, it is

not necessary to consider this time as a function of the time to run a given application - the neural network is not a terminating "program".

- Communication to the Outside World: SpiNNaker needs to be attached to a Host PC to load the neural application and interact with the user. Typically, the Host PC would be a normal PC, necessitating some way of connecting SpiNNaker to it. Every chip has an Ethernet interface, but given that only one or at most a few chips would connect to the Host, there must be a protocol bridging the Ethernet communication (between the Host PC and Host-connected chips) and packet-based communication (among the CMPs).

- Run-time Interactive Support: Once the application is running, we need to interact with the system to examine the state of hardware devices and the application running on the chips. A common communication language using small packets to interact with each chip's monitor processor is needed.

6.6 Summary

CMP technology is expected soon to take over from conventional computing system design for its better performance, scalability, power-efficiency and fault-tolerance. A multi-CMP system provides computing power to run very complex scientific applications beyond the scope of conventional high-end workstations, or a cluster of them, in a much more cost-effective manner using process/thread level parallelism. Such a system can be viewed as a PC cluster, each a multi-core workstation interconnected over an asynchronous network performing parallel distributed computing. This brings in new challenges for the use of such systems, such as intra-CMP and inter-CMP application-level synchronization and the user's interaction with applications etc. These challenges require configuring these systems in a way to support efficient inter-process communication, system-/application-level run-time interaction, application loading, and fault-handling etc. The purpose is to protect the user from architectural complexities and present the system as a one computing unit as if the user were working on a single PC. Various multi-CMP systems have been reviewed to investigate the manner these issues are being handled. SpiNNaker is a novel multi-CMP system designed with a specific purpose. It has been especially architected to support

large-scale neural simulations using low-power embedded processors, networked with an asynchronous interconnect. Its architecture is quite different from other SoC-based computing systems such as the IBM's Blue Gene and Cray's XT3. Though faced with similar configuration challenges, SpiNNaker needs a different way to tackle these problems due to its unique architecture. This provides a motivation to research these issues in the context of the SpiNNaker computing system to devise a mechanism suitable for multi-CMP computing systems designed following SpiNNaker's architectural principles.

Chapter 7

SpiNNaker Configuration Process

7.1 Introduction

Chip-Mutliprocessor (CMP) architecture uses multiple simple and low-power processing cores providing the best choice for the present day's applications with process- or thread-level parallelism. The CMP follows a hierarchical partitioned design approach with replicated modules using short and fast communication media as compared to those of conventional systems [BKM+00]. Once a CMP with an on-chip networking and routing mechanism is produced, a number of such chips can be connected together to form a scalable multi-CMP high-end server machine. This approach is being adopted by most high-end machine vendors such as IBM in its Power4-based systems [TDF+02], Compaq in its Piranha-based systems [BKM+00], Sun in its next generation Niagara T2 UltraSPARC system [AKB+07b], and Cray in its XT series machines [Inc05b, AKB+07a] etc.

As explained in Chapter 6, multi-CMP system design brings with it a number of application-level challenges such as system-wide synchronization, system-level configuration, an application loading mechanism, debugging, and run-time user-application interaction. We also examined a few SoC based multi-chip systems for studying configuration strategies adopted to understand how these challenges have been handled in these systems. The chapter ended with a conclusion that we need a novel configuration and application loading mechanism for the SpiNNaker multi-CMP system because of its novel architecture. As described in Chapter 4, SpiNNaker is a scalable multi-CMP system for running complex neural simulation with inherent process- and thread-level parallelism. Simulating large, biologically-realistic neural networks is an excellent candidate application

for distributed processing systems: indeed, the consensus in the modelling community is that it may be necessary to use dedicated hardware with architectures more closely similar to the brain for large-scale neural modelling within realistic resource limitations [JSR+97]. It is efficient to simulate a spiking neural network as an event-driven real-time application [JFW08], a model quite different from typical parallel applications and more akin to embedded applications [Kop97]. A system for neural network simulation will be, correspondingly, architecturally different from parallel systems designed mostly for general-purpose computing. SpiNNaker adopts event-driven models of computation, and its boot-time configuration can make fewer assumptions about the initial state of the system than "conventional" parallel multiprocessor systems.

SpiNNaker provides no side-band communication channel for boot processes: the system boot must use the same communication fabric as the application. All processors on the chip are identical; there is no dedicated processor hard-wired or preconfigured to run the boot process. The task, therefore, is as follows: It is necessary to configure a symmetric massively-parallel system using only the resources available at run time, even though the functionality of these resources themselves depends upon having been configured. The configuration process must do this efficiently and without contention, even though individual processors have only local state information available, i.e. the system can use no global state information to configure itself. The system must somehow break symmetry, assign and load memory resources, configure communications, and start up the processors, while balancing concurrency and resource contention for maximum efficiency. Where previous solutions [HBB+05, Inc05b] have typically used side-band communications or dedicated preconfigured resources, SpiNNaker confronts the challenge of configuring an isotropic undifferentiated parallel processing system head-on. One approach would be to make no assumptions about the application and consider it as a problem in general-purpose computing, leading to a set of standardised, generic configuration techniques. However, since numerous studies indicate that parallel processing works best with specific applications that have inherent parallelism, it seems reasonable to design parallel systems around a target application, whose boot process could be correspondingly specialised.

A flexible and efficient boot loading of distributed applications is an essential support process for the SpiNNaker multi-CMP massively-parallel system organized over a homogeneous communication fabric [KNR+09]. This chapter presents

the SpiNNaker multi-CMP system configuration process and a run-time application support mechanism. The process has been designed specifically for the SpiNNaker massively-parallel multi-CMP system. However, the techniques used may also be useful in other multi-CMP systems based on linking functionally-independent CMPs over an asynchronous network; a concept taking over the high-end computing industry. Some experimental results are also presented at the end to justify our claimed objectives.

7.2 SpiNNaker Boot-up Process

The SpiNNaker multi-CMP system is configured in two phases. In the first phase, the processors run the Boot ROM code in batch mode to test processing cores and on-chip peripherals independently. In the second phase, the configuration process follows the SpiNNaker event-driven model to configure the whole system at run-time. The two phases are explained in the succeeding sections.

Phase I - Chip Level Configuration

Each SpiNNaker CMP must perform basic power-on testing and initialization based on the instructions in the Boot ROM. Initially the ARM968 Tightly Coupled Memory (TCM) is disabled and the processors access a high-interrupt-vector in Boot ROM at power-on reset. At this point, all the processors run at very low frequency (10 MHz) to conserve energy. After testing and enabling the TCM, the boot-strap code is copied to the local instruction memory (ITCM) of each processor for better performance. The process of copying the code from the boot ROM simultaneously by the 20 processing cores causes a bit of contention. However, once the code is in the local memory each processor can run it independently. As part of the initial boot-up, each processor follows the following sequence of actions (Figure 7.1).

- Each processor performs a power-on self-test (POST) on the processing core's peripherals, i.e. the Communication Controller, the Vector Interrupt Controller, the Timer and the Direct Memory Access (DMA) Controller, to identify any fault. For every processing node to function properly, it is important that all its peripherals should be functional. In case of a failure, the processing node puts itself into a sleep mode to avoid any disruption to the CMP.

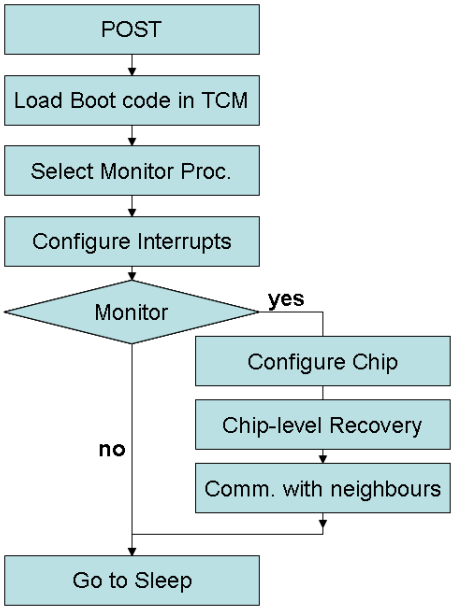

Figure 7.1: The SpiNNaker Boot-up Process - Phase I.

- Each processor initializes its peripherals to work on a default setting. The Communication Controller is initialized to send and receive packets, the timer is initialized to generate an interrupt after every millisecond to be used for synchronizing system-wide activities with millisecond granularity, the Vector Interrupt Controller is configured to pass on basic interrupts such as the packet received, error interrupts and the timer interrupt, while the DMA is set-up to carry out DMA read/write operations and to raise any error interrupt.

- Unlike most CMP systems, the monitor processor is not hard-wired at design time as this might introduce a single point-of-failure. In the SpiNNaker CMP, the monitor processor is selected from the 20 on-board processors at run-time with the help of the boot-up instructions. To this end, all healthy processors compete to access the System Controller across the System NoC. The System NoC's arbiter allows only one processor to access the System Controller in case of an access contention, and the first processor to gain this access is selected by the System Controller as the monitor processor. The system controller writes the ID of the selected monitor processor to one of its registers, which all other processors look up to identify the monitor.

- All processors inform the System Controller of their state. For this purpose a specific register has been included in the System Controller. Each processor, after successful test and initialization of its peripherals, sets the bit corresponding to its ID in this register. This information provides the monitor processor with the state of other processors. This information is used later for chip management, application configuration and reporting the state of the chip to the Host PC as explained later. Additionally, the system controller maintains the state of other chip resources which can be queried from the outside with the help of the monitor processor.

At this stage all processors, except the monitor, go into sleep mode. The monitor processor, which is now managing the chip, performs chip-level testing and initialization. The following sequence is adopted by the monitor processor:

- It switches the chip clock to a faster frequency i.e. 200MHz. This feature is controlled with the help of the Phase-Lock-Loop (PLL) control register in the System Controller.

- It performs detailed device-level tests and writes the state of chip-resources to the System Controller for later reporting to the Host PC outside the SpiNNaker system.

- It initializes the chip's resources such as the PL340 SDRAM Controller, Router, and the System Controller etc. At this moment, the router can route only the Nearest Neighbour (NN) packets as the other two types of packet (Multicast (MC) and Point-to-Point (P2P)) require relevant routing tables to be configured.

- It establishes if a PHY (Ethernet Physical Layer Module) is present. If so, this chip may be the one (or one of those) connected to the Host PC. The monitor processor initializes the PHY, and if a live connection to the Host PC is detected, it initializes the Ethernet Interface to start receiving frames. If there is no live connection, it does only the basic initialization of the Ethernet Interface and PHY. In this case, a detailed Ethernet Configuration is performed later, on receipt of a "connection-up" PHY-Interrupt event. If, however, the monitor processor does not detect a PHY, it disables the Ethernet Interface to save power.

- It carries out basic functional tests such as sending a test NN packet to itself for testing correct functionality of the on-chip Communication NoC, sending a loop back Ethernet frame to test the Ethernet functionality, and testing for DMA read/write operations.
- It performs a chip-level recovery to restore any faulty chip components using embedded recovery routines. It can also reset or disable the clock of a malfunctioning application processor to restore or isolate it as part of chip-level recovery.
- It tests connections to neighbouring chips by broadcasting a "Hello" NN packet to all neighbours, marking any faulty links or neighbours. This initiates a "nearest neighbour diagnostic process" to identify any dead chip and to recover, reset or isolate it.
- It configures the Watchdog Timer's interrupt and reset mechanism to reset the monitor processor or the whole chip in case of a non-responsive monitor processor as part of chip-level recovery. The mechanism will be explained later in Chapter 8.

After this process, the monitor processor also enters sleep mode putting the chip in listening mode waiting for internal (from Timer or Watchdog Timer etc) and external events (packets or Ethernet frames etc) notified as interrupts.

Phase II - System Level Configuration

In this phase, the configuration process runs as an event-driven application under the control of the Vector Interrupt Controller. At the chip(s) connected to the Host PC, the PHY generates connection-related events and the Ethernet Interface generates a "frame-received" event as an interrupt to the monitor processor. On all chips, the Monitor processor's Communication Controller generates a "packet-received" event when a message arrives from a neighbouring chip. Each interrupt triggers a different Interrupt Service Routine (ISR) to run the code for the related configuration job before putting the monitor processor to sleep again as shown in Figure 7.2. These two different event-driven processes, i.e. the packet-received and the Ethernet frame-received, use two separate communication protocols. The two protocols are explained in Appendix A and Appendix B of this thesis. The

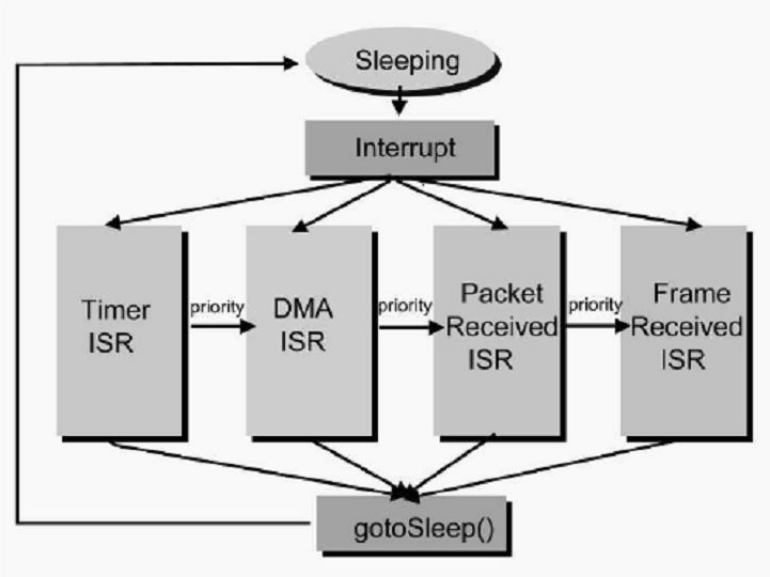

Figure 7.2: Event-Driven System-level Configuration.

monitor processor on the Host-connected chip translates between the two protocols, converting the Ethernet frames it gets from the Host PC to packet-based messages, then issuing them either as broadcast or chip-specific NN packets. Besides these two interrupts, timer- and DMA-interrupts are also used in this process to support a process-timeout and memory-transfer operations as part of system-level configuration process. Figure 7.3 shows the flow of phase II of the SpiNNaker configuration process.

From Phase I, each chip should receive a Hello message from all its neighbours within a certain time. If a given link times out, the monitor processor activates a "neighbour diagnostic" routine which establishes the problem and tries to cure the faulty chip. The process will be explained in Chapter 8.

Following any necessary time for the neighbours diagnostic process, the chips start executing event-driven system-level configuration. This, the main component of system-level boot-up, operates as follows.

- The chip(s) connected to the Host PC notify the system readiness to the Host PC by sending a "Hello" frame, indicating that the system has now completed its Phase I process and the NN diagnostics.

- The Host PC nominates one of the Host-connected chips to be the "reference chip" i.e. the one to bear the reference address (0,0) and notifies it of the

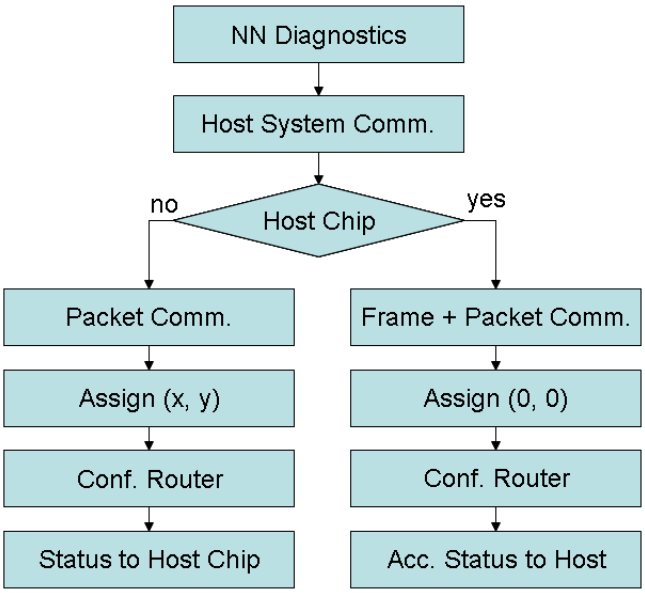

Figure 7.3: The SpiNNaker Boot-up Process - Phase II.

number of chips in the system. It then instructs the chips to assign unique chip addresses dynamically. The origin chip broadcasts its address with the size of the system. The adjacent chips compute their addresses by size modulo addition to find a relative address (x,y) in a 2D SpiNNaker torus, and broadcast their address forward. This process continues outward to cover the whole system.

- Each chip configures its P2P table based on the logical location of the chips based on the size of the system to perform default P2P routing. (The application can later modify these tables according to the system-level configuration).

- The Host PC requests the system state. Each chip reports its state to the origin chip using P2P packet(s). The origin chip accumulates these states and reports the result to the Host PC.

- The Host PC loads a micro-kernel and run-time configuration code to the chips using a flood-fill mechanism as explained in Section 7.3.

- The Host PC loads the application to the chips using the flood-fill mechanism as explained in Section 7.3.

- The Host PC computes the MC routing tables with the help of a user-interface application as per the user's application model according to the chips' state.
- The Host PC configures the routing tables for each chip with the help of P2P packets as computed in the previous step.
- The Host PC instructs the monitor processors to activate the application processors to load the application to their local memories and start running the application.
- The Host PC interacts with the system either to carry stimuli/responses on behalf of the application or to interrogate the state of the system or the application.

The protocol follows a set of instructions defined in Appendix B, passed between the Host PC and the SpiNNaker system using Ethernet frames.

7.3 The Application Loading Process

A typical SpiNNaker application will need to load neural support data into each SpiNNaker chip's memory before it can start its execution. The data may include the initial state of the application, neural network mapping, inter-neuron connectivity information, and synaptic information. In addition, a micro-kernel and the utility functions library for the monitor processor may also need to be loaded into each chip. We need an efficient way to load this data in the minimum possible time.

An efficient and fault-tolerant mechanism has been devised to load the application and data into the chips. During the inter-chip flood-fill process each CMP uses NN packets to send a 32-bit word of data at a time to its 6 neighbouring chips. The receiving chips store the data and broadcast it on to their neighbours. A pipelined "wave" of data thus flows from the Host-connected chip(s) to the whole system. To maintain data flow control, we have devised a special instruction set to be used with NN packets, the instructions are included as Appendix A to this thesis. The SpiNNaker address space 0x0F800000-0x0FFFFFFF has been reserved for specific instructions in the NN packet's address field. The monitor processor interprets an NN packet with an address in this range as a particular

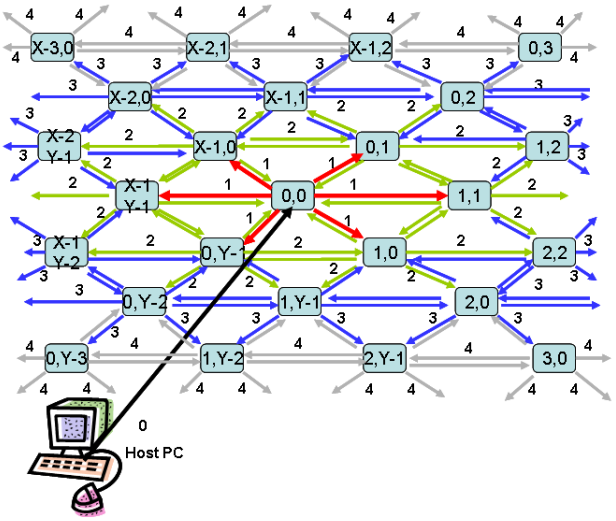

Figure 7.4: Selective Forward Flood-fill.

configuration process instruction or a response to such an instruction. This protocol covers instructions to serialize the data, control the flow of data, requesting missing bits of data and various other configuration instructions. We have devised various flood-fill mechanisms such as "broadcast" or "selective forward multicast" which gives various performance vs. robustness tradeoffs. Figure 7.4 shows one such mechanism for flood-filling the data in the SpiNNaker system. Figure 7.5 shows the sequence diagram of the flood-fill mechanism which works as follows:

- The Host PC loads the application and data as small (1-8K) data blocks to the system. It sends a data block along with its size, start address, and block-level (Cyclic Redundancy Check (CRC)) checksum to the Host-connected chip(s) one data-block at a time using Ethernet frames.

- Each Host-connected chip performs a checksum test on the block to ensure its correct receipt. If an error occurs, it requests the Host PC to resend the block.

- The chip(s) connected to the Host PC send an instruction through an NN packet to indicate the size of the block to be transmitted and the starting location to store it in the SpiNNaker chip address space.

- The system inserts the physical address of the word into the routing key of the NN Packet to help serialization and duplication control.

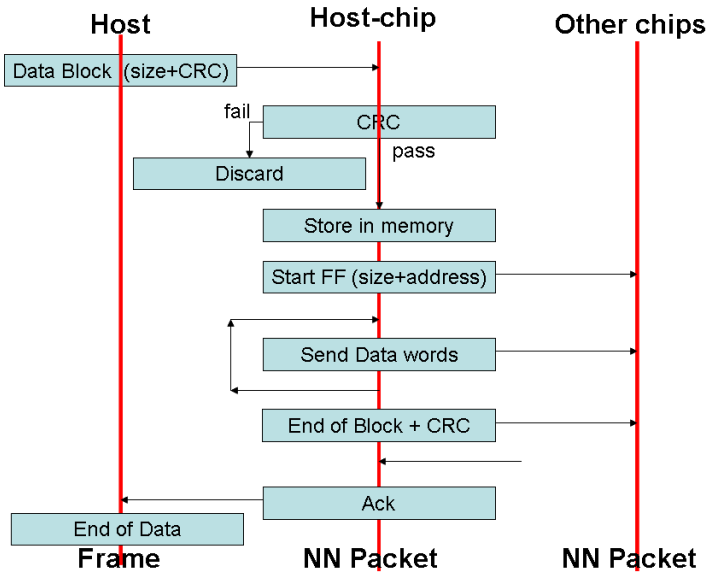

Figure 7.5: Flood-fill Process - Sequence Diagram.

- Each neighbouring chip dedicates a memory space in its data TCM equivalent to the size of the data block being received. It also initializes a "word received" bitmap with a number of bits equal to the number of words in the block. On receipt of a packet, it will inspect the corresponding bit against the routing key (address) of the data in the packet. If a word has already been received, it will neither be stored in the memory buffer nor be transmitted further. Missing words (zeroes in the bitmap) can be requested from neighbours at the end of the flood fill.

- The last NN packet of data contains an instruction indicating the end of the block along with the block-level checksum in the payload.

- If the block passes the CRC test, the receiving chips load it into the specified location in the memory address space.

- If the received block contains the application, the chip loads it into local memory, ready to be executed on receipt of an instruction from the Host PC.

- The flood-fill process ensures that each chip receives the transmitted word at least twice during the flood-fill process, ensuring data delivery to each chip.

- At the end of the flood-fill process, the Host PC requests the state of each chip along with blocks received. At this stage, the chips can request missing blocks from each other or from the Host PC.
- During the "neighbouring chip recovery" process, chips surrounding the faulty one can transport the data locally to the recovered chip.

7.4 Configuration Issues

Chapter 6 gives some important application-level issues in a multi-CMP system which the configuration process needs to look into to present a system-wide integrated view of the system to the user. Table 7.1 presents a comparison of various approaches used in the three architectures to handle these issues.

Table 7.1: Configuration Issues Handling Approaches.

S/ No	**Configuration Issues**	**System**	**How Handled?**
1	Chip management	BG(L)	By a dedicated service processors running BLRTS kernel.
		Cray XT4	Jointly by a service processor on the communication chip and the main Opteron processor with the help of an operating system.
		SpiNNaker	By a monitor processor on each chip.
2	Selection of monitor proc.	BG(L)	A service processor is hardwired in each CMP at the design-time.
		Cray XT4	A service processor is kept for initial boot-up in the communication chip. Later on, the monitoring is done by the Opteron main processor.
		SpiNNaker	Selected at configuration time with the help of a random selection process to minimize single points-of-failure.
3	Chip-level testing/initialization	BG(L)	Performed with the help of boot-up code loaded externally using JTAG and a dedicated Ethernet, attaching each CMP to a service node.

		Cray XT4	Performed externally through a separate management network connected to each Compute PE which allows it to access all registers and memories.
		SpiNNaker	Done with the help of boot-up code in each chip's Boot ROM. Each processor initializes its peripherals while the chip-level shared resources are tested and initialized by the monitor processor.
4	System-wide configuration	BG(L)	Performed externally with the help of a global operating system using the Ethernet network. The service nodes communicate with the operating system on the service processor in each CMP to integrate the system and manage it.
		Cray XT4	Initial configuration is done with the help of a management network and service processor on the communication chip, later on it is done using the operating system on the Compute PE's Opteron.
		SpiNNaker	Done with the help of the same homogeneous asynchronous network used for the application. The Host PC is attached to only one or a few chips to interact with the entire system.
5	Application loading	BG(L)	Initially a small boot-loader is loaded externally to each chip's local memory to configure the Ethernet network and understand application loading protocol. Following this the application is loaded into each CMP's local memory using Ethernet network and service processor on each chip running boot-up code.

		Cray XT4	It is done using the management network. The application can also be accessed from the central file system using the system-interconnect, after the initial configuration.
		SpiNNaker	The system-wide flood-fill process to load the application and data using the same inter-CMP interconnect to be used by the application. No global shared file system or storage accessible to the CMPs.
6	Network configuration	BG(L)	With the help of boot-code loaded to each node, each chip is assigned a torus interconnect address and an IP address. Following this, the inter-CMP torus is configured to carry packets.
		Cray XT4	Done externally using the management network and the service processor. Both the networks i.e. the Ethernet and the system-interconnect are configured initially.
		SpiNNaker	The network does not require configuring for initial configuration and the flood-fill processes. Later it is configured externally with the help of the same interconnect.
7	Inter-process communication	BG(L)	With the help of (256-byte) packets on the torus interconnect.
		Cray XT4	With the help of (64-bytes) packets on the torus interconnect. Shared memory OpenMP is used for messages within a node. A global shared file system can also be accessed through the DMA controller and the system-interconnect.
		SpiNNaker	With the help of (5-9 byte) packets. No shared system-wide file system or storage. Shared memory message passing within a CMP.

8	System-wide synchronization	BG(L)	Performed with the help of an external network, and later by the torus interconnect using a global operating system and DBMS interacting with the BLRTS kernel running on the service processors.
		Cray XT4	Performed using the on-chip and global operating systems on the service nodes communicating over the system-interconnect and a separate management network.
		SpiNNaker	Can be achieved by broadcasting NN packets around the system on the instructions of the Host PC.
9	System monitoring/ management	BG(L)	With the help of external network using service nodes. The state of the system is maintained using a DBMS and a global operating system collaborating with the BLRTS microkernel running on each CMP.
		Cray XT4	With the help of a management network connecting all the compute nodes. A global system-wide operating system gives an integrated view of the system with a graphical view for easy system management.
		SpiNNaker	Done by the Host PC using the same Communication Network used by the application. A graphical user interface is intended to give an integrated view of the whole system using the system interconnect and the Host-system interconnect.
10	User-Application interaction	BG(L)	With the help of external network and the global operating system.

		Cray XT4	Initially with the management network, but later with both management network and system-interconnect as the service nodes are directly attached to the system-interconnect.
		SpiNNaker	With the help of the user interface on the Host PC using the same system interconnect.

7.5 Evaluation Work

7.5.1 CMP Boot-up

The SpiNNaker multi-CMP configuration protocol and the application loading process for the SpiNNaker system have been developed using the ARM RealView Development Suite to generate a loadable Boot ROM binary image for ARM968E-S processing cores. We tested the code for its functional correctness, performance and scalability on the cycle accurate system-level TF-AV(CA) model of SpiNNaker described Section 5.6.4, and a top-level Verilog behavioural model. Phase I of the boot-up process is straightforward batch mode code loaded by each ARM968E-S to its ITCM from the Boot ROM and executed independently for processing-node initialization, and later by the monitor processor to initialize the chip. Table 7.5.1 shows the boot-up time as the number of ARM968 CPU cycles (with a 200MHz target clock rate as in the SpiNNaker CMP) on the SystemC system-level model. Code execution time does not depend on the number of CMPs in a multi-CMP SpiNNaker computing system since it runs concurrently on all the chips. The time is slightly affected by the number of processing nodes in a CMP as they share the same Boot ROM to read and copy the boot-up code to their ITCM which introduces System-NoC access contention. Column 3 shows the number of ARM968 CPU cycles (at 200MHz) used to execute chip-level boot code (except CRC table initialization which takes 1.09ms), while column 4 shows the simulation speed on the host PC (an Intel Core2 duo 1.6GHz, 2GB RAM running WindowsXP).

Table 7.2: CMP-level Boot-up Process Time

No. of CMPs	No. of Procs. per CMP	No. of CPU Cycles	Sim. Time (sec)
1	1	275887	6.00
1	3	275990	8.28
5	3	275990	41.60
9	3	275990	77.86

7.5.2 Application Loading

For phase II and the application loading process, we carried out our experiments on the SpiNNaker TF-AV(CA) system-level model (Chapter 5, Section 5.6.4) for functional correctness. To test the performance and scalability of various proposed application loading algorithms, experiments were carried out with the help of a high-level event-driven network simulator [KNJ+08b, KNJ+08a] for a multi-CMP SpiNNaker computing system with various sizes (up to 64Ki CMPS).

We experimented with the following distribution algorithms for the application loading flood-fill process:

- 2Msg Forward (Figure 7.6a): the monitor processor sends messages to the adjacent neighbours on the links diagonally opposite to the one it received the packet from i.e. in to the chips (x+1, y) and (x, y+1) in 2D torus configuration of the SpiNNaker multi-CMP system. These are the minimum number of packets required to perform an efficiently-pipelined flood-fill distribution of the application. Each packet needs 1 router cycle i.e. a chip will use two router cycles to forward the message.

- 3Msg Forward (Figure 7.6b): this mechanism operates like the 2Msg forward, except that the packet is sent to the diagonally opposite link and its two adjacent links. The process is slower than the 2Msg Forward as it takes three router cycles to send packets to three neighbours, however, it is more reliable as it ensures that each chip receives a packet at least twice, which is not the case with the 2Msg Forward algorithm.

- 5Msg Forward (Figure 7.6c): this process causes a CMP to send a copy of the packet to all neighbours except the one it received the packet from. This requires 5 router cycles as each neighbouring chip is sent the same packet individually.

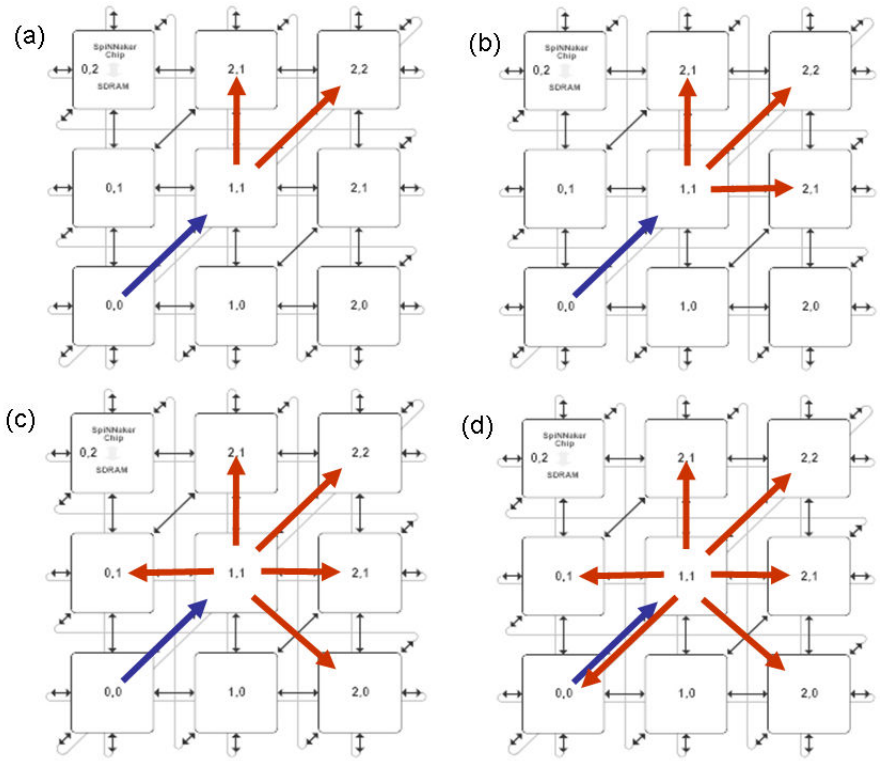

Figure 7.6: Application Loading Process - Flood-fill Approaches [KNJ+08b, KNJ+08a, KNR+09].

- Broadcast (Figure 7.6d): the monitor processor on each chip uses the NN broadcast feature to send a copy of each packet to all its nearest neighbours using only 1 router cycle.

- rndXX: like 2Msg Forward but adds (XX%) probability to send packets to the other neighbours i.e. a packet is forwarded to either 2, 3, 4, or 5 neighbouring chips selected randomly to avoid fixed inter-chip traffic congestion. We tested probability values 25% (rnd25), 50% (rnd50) and 75% (rnd75).

We implemented the proposed algorithms and tested them on the SpiNNaker system-level TF-AV(CA) model connected to an external application using TCP/IP sockets to flood-fill the application and data into the system model. The timing computed for the Ethernet frame and packet delays on a 9xCMP SpiNNaker TF-AV(CA) model was used in the high-level SpiNNaker network simulation to acquire approximately the same accuracy. The high-level model simulates the behaviour of the inter-CMP asynchronous interconnect capturing

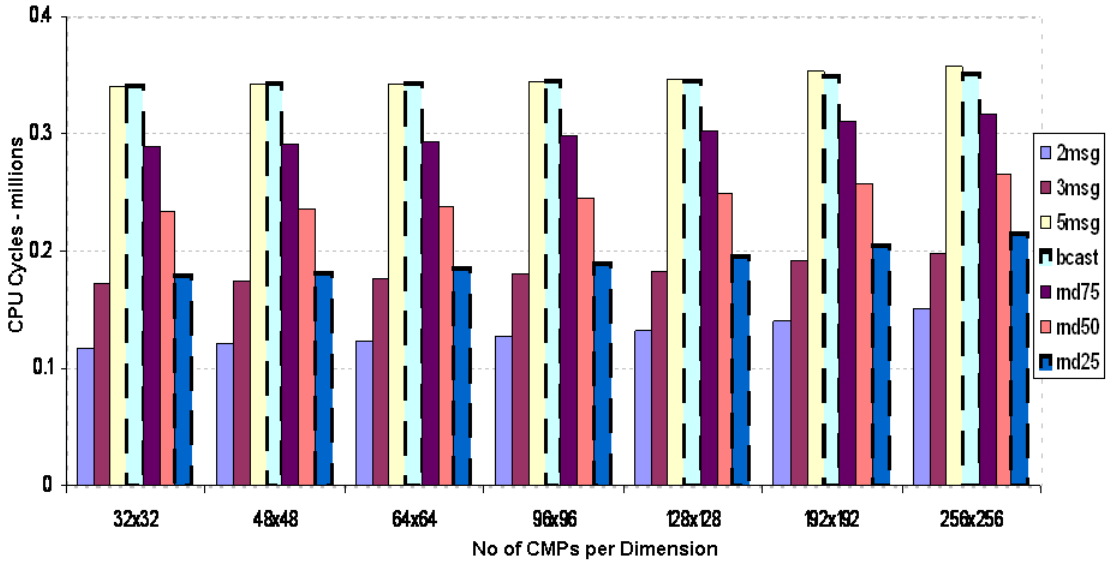

Figure 7.7: Application Loading Process - Impact of System Size (10-Kbyte Date with 1 Ethernet Connection to the Host PC) [KNJ+08b, KNJ+08a, KNR+09].

the traffic load, network congestion, packets dropped at each node due to congestion, and inter-chip link blockage etc. We tested the system using 1, 2 and 4 Ethernet connections to the Host PC connected to the CMPs at location (0,0), ($X/2$,$Y/2$),($X/2$,0) and (0,$Y/2$) in the 2D SpiNNaker torus as explained in Chapter 4. We tested various network sizes, all of them square, ranging from 32x32 to 256x256. For these experiments, we used an error-free configuration of the network as the purpose was to analyse the configuration process for performance and scalability.

Impact of System Size

Figure 7.7 shows the results of our experiments conducted with high-level SpiNNaker network simulation. We flood-filled a fixed amount of data (8KB Application + 16KB data = 24KB) on the SpiNNaker system with 32x32, 64x64, 128x128, and 256x256 chips.

The results show that the 2Msg Forward mechanism is the best in performance, followed by rnd25, 3Msg Forward, broadcast, rnd50, rnd75, and 5Msg Forward respectively. Though in the broadcast mechanism each chip takes only

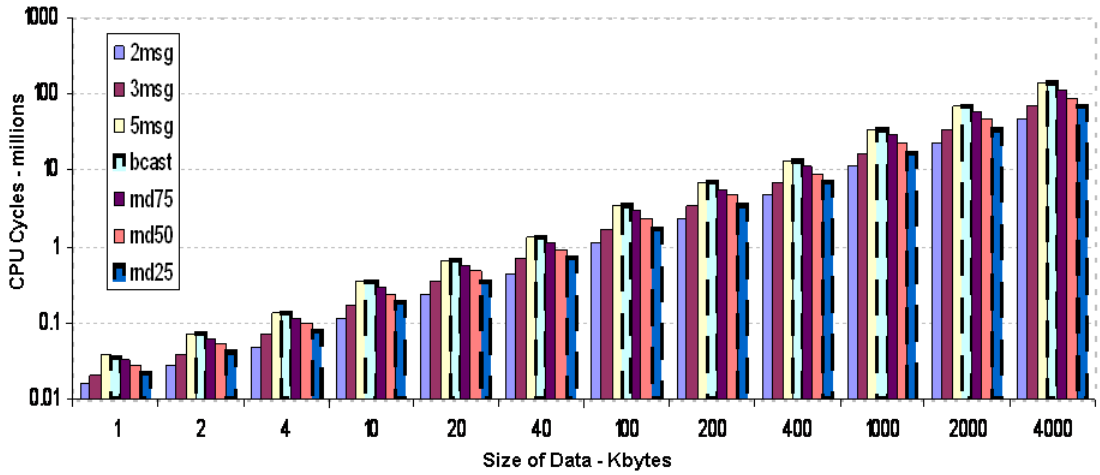

Figure 7.8: Application Loading Process - Impact of Data Size (32x32 Nodes System with 1 Ethernet Connection to the Host PC) [KNJ+08b, KNJ+08a, KNR+09].

one router cycle to send a copy of the packet to its six neighbouring chips, it does not give the best performance. This is due to the network congestion caused by broadcasting the packet to all chips by all chips, which reduces the network performance as the recipient router runs out of buffer space having to wait for the monitor processor to clear the packets. Each chip gets 6 copies of the same packet which the monitor processor has to receive before discarding the duplicates. 5Msg Forward is the worst for taking the maximum router cycles (5) for sending the packet to neighbours individually, and introducing almost the same amount of congestion as with the broadcast. The 2Msg Forward algorithm is the fastest method, however, it is not very robust as the algorithm does not ensure that each chip receives duplicate packets which may prevent all the chips in a pipeline from receiving the data if some chip in the start does not get a packet due to a chip or link error. For all distribution policies, the application load time in a large multi-CMP system is virtually independent of both the number of Ethernet connections and the system size. The only relevant parameters are the amount of data and the distribution policy. This is because of the perfect pipelining of the packet distribution mechanisms.

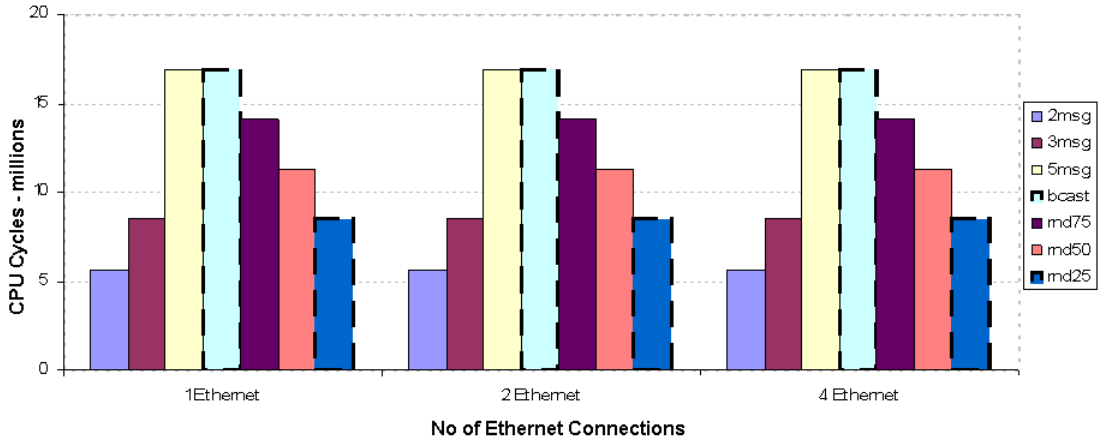

Figure 7.9: Application Loading Process - Impact of Ethernet Connections (10-Kbyte Date on a 256x256 Node System) [KNJ+08b, KNJ+08a, KNR+09].

Impact of Data Size

Figure 7.8 shows the impact of data size on the flood-fill process of loading the application and data into the SpiNNaker multi-CMP system. We experimented with varying sizes of data (application+supporting data) i.e. 8KB+16KB, 16KB+32KB, and 32KB+64KB of application plus supporting data into the SpiNNaker system comprising 128x128 SpiNNaker CMP. We used only one Ethernet connection between the system and the Host PC.

The results show that the process of application loading is dependent on the size of data to be loaded, however, the total time taken (20 ms for 32KB+64KB data) is not that high. The efficiency of the distribution algorithms is the same as that explained in the previous experiment for the impact of system size. These are the expected results as we are required to send more Ethernet frames and thus more packets in the pipeline to load larger amount of data.

7.5.3 Impact of Ethernet Connections

Figure 7.9 shows the results of the simulation conducted to evaluate the impact of the number of Ethernet connections between the Host PC and the multi-CMP SpiNNaker system. We used a SpiNNaker system with 128x128 CMPs to load a 32KB application with 64KB of supporting data (as expected to be loaded in the SpiNNaker system without synaptic data).

Surprisingly, the flood-fill process does not depend substantially on the number of Ethernet links to the Host PC in a large multi-CMP system, exhibiting negligible performance gain with 4 Ethernet links. This is because of the efficient Communication Network and the pipelined flood-fill application loading process. For a large system size and large amount of data, the improvement gained by multiple Ethernet connections is not pronounced. However, the redundancy of connections to the Host PC provides better reliability by reducing single points-of-failure for application loading. This will be further explained in Chapter 8.

7.6 Summary

Configuring a multi-CMP system at run-time to load and run a variety of applications is a challenge in an asynchronous distributed structure. The SpiNNaker massively-parallel computing system is a novel multi-CMP system with many in-house designed components. SpiNNaker has been designed to support most spiking neural networks simulations. To achieve this objective, SpiNNaker presents a "blank-slate" view to load a variety of neural applications at run-time and configuring the system according to the application. A novel configuration process has been devised as part of this research to configure a multi-CMP SpiNNaker system dynamically and load the application at run-time. Experimental results verify our claims for the process to be highly efficient and scalable. Although the process has been devised for the SpiNNaker multi-CMP system, it is expected to be useful for other multi-CMP systems.

Chapter 8

Fault-Tolerance

"Our brains keep working despite frequent failures of their component neurons, and this 'fault-tolerant' characteristic is of great interest to engineers who wish to make computers more reliable,..." S. B. Furber [New06]

8.1 Introduction

Natural artifacts are, generally, more reliable than their man-made replacements. The human nervous system is an engineering masterpiece which human engineering is still not approaching in terms of efficiency, power consumption, heat dissipation, or robustness. Despite using massive numbers of small processing units which form complex interconnections, the human brain is a very fault-tolerant "real-time" system. Nature uses the principles of resource redundancy, self-organization, adaptability, local reconfiguration, and autonomous local recovery to maintain reliability in the nervous system. Computer science, during its 60 years of exponential evolution, has yet to achieve this level of reliability in its engineered components. There is a need to reconsider our engineering methodology to align with that of nature in order to design reliable components, otherwise catastrophic incidents and expensive industrial losses may continue, such as the deadly accident of space-shuttle Challenger due to component malfunction.

One of the objectives of the SpiNNaker project is to learn the principles of designing fault-tolerant systems from biology. Our aim is to use the known to discover the unknown i.e. to use engineering techniques from nature to design a fault-tolerant computing system, and then to use this platform to explore further and to improve our hardware. SpiNNaker has been designed to simulate

a part of the nervous system in biological real time. The biologically-inspired neural networks have to be temporally correct within a required temporal granularity (a millisecond in case of functional simulation of spiking neuron to capture spike dynamics and the learning behaviour of a neural network) [Me98, DA01]. The SpiNNaker computing system running such a simulation functions like a "real-time system" and has to simulate the both "logically and temporally correct" [Gre08] behaviour of the simulated neural network. This means that besides producing functionally correct results, the SpiNNaker computing system must perform these functions within an explicit timeframe or the results may not be realistic. A real-time system has to be fault-tolerant or it fails to perform in real-time [Kop97]. Conversely, a fault-tolerant system qualifies as a real-time system as all the fault-handling functions have to meet some temporal deadlines to maintain the usefulness of the system [Gre08].

The SpiNNaker computing system depends on its configuration process to make use of its specially designed fault-tolerance features. On the other hand, the SpiNNaker configuration process has been devised to be as fault-resilient as possible so as to bring the system into functional state despite component-level malfunction up to a certain degree. This chapter introduces the fault-tolerance features of the SpiNNaker configuration process proposed in this thesis. The chapter begins with a review of some fault-tolerance concepts in the context of real-time computing systems, followed by a description of the hardware support in the SpiNNaker computing system to facilitate fault-tolerance. At the end, the chapter explains how the configuration process uses SpiNNaker's fault-tolerance features to make it a reliable computing system.

8.2 Fault-tolerance In Computing Systems

System fault-tolerance requires the system to be able to detect an error and restore itself autonomously to a reasonable operational state, or at least to notify the problem to the user (if possible) [RLT78]. "A system is considered to be fault tolerant if it can continue to operate in the presence of failures (albeit, with perhaps degraded performance)" [Gre08]. While devising a completely fault-tolerant system and able to recover from any real-time failure is difficult, certain faults can easily be trapped and corrected, restoring the system to at least a minimal level of functionality. One of the central issues related to fault-tolerance

in real-time systems is the *cost of recovery*: the time and resource utilization needed to correct a fault. Sometimes the recovery time may be so long that the source error propagates, causing further undesirable faults [Gre08]. In that case, it may be more practical to isolate a faulty component than to try to restore it. However, we cannot isolate a critical component upon which other components depend. We can minimise the risk of such points of failure through resource redundancy or alternative workaround mechanisms.

Before discussing fault-tolerance techniques in a computing system, it is worthwhile to introduce some basic terms. "When the behaviour of a system deviates from that which is specified for it", this is called *failure* [RLT78]. System failure is an event that affects the reliability of a system. The term *error* means "an incorrect state of the system" [RLT78], while a *fault* is "the mechanical or algorithmic cause of an error" [RLT78]. Undesirable events are called *hazards* which can cause defects or faults which eventually lead to failures. In the worst case, failures can propagate to cause a catastrophic breakdown or *mishap*. The following relationship shows the inter-relationship of these terms in a computing system [Gre08].

$$\text{fault} \Longrightarrow \text{failure} \Longrightarrow \text{hazard} \Longrightarrow \text{mishap}$$

Real-time systems can be divided into two types: *hard-lined* and *soft-lined*. A hard-lined real-time system that misses its time constraints may bring a system into a mishap, examples include safety equipment and aircraft control systems, while in soft-lined systems the time constraints are desirable but do not cause a severe failure [Kop97].

Fault-tolerance in a computing system may involve all or most of the following techniques.

- Fault Detection: The purpose of fault detection is the prevention of a system failure by recognizing when a fault may be about to happen [RLT78]. A computing system can have diagnostic mechanisms which run periodically to identify potential problems or devices can be designed to detect faults at run-time and report these to the user or the application.

- Fault Treatment: A detected error may be only a symptom of the fault that caused it. We need to find out the exact nature of the problem and its potential effects before taking measures to avoid failure. Many faults may cause a similar error and sometimes ignoring the fault and continuing

to provide the current service may be the best alternative [RLT78]. The fault treatment process may require a detailed assessment to determine the extent of the damage (though this is not possible in some real-time systems).

- Error Recovery: This is a process which attempts to correct, mitigate, or contain failures so that no further problems are introduced [Gre08]. Some options to error recovery may be (but are not limited to):

  1. Replacement: i.e. replacing a component with a spare one. This relies on component redundancy.
  2. Reconfiguration: i.e. to reconfigure a faulty system (or a part of it) to bring it out of a faulty state [RLT78]. This involves changing the failed component's internal configuration and/or its interactions until a desired functionality is restored [Gre08]. It does not guarantee full recovery, however, it is one of the most viable options in use in self-adaptive hardware.

- Fault Isolation: i.e. to attempt to isolate the fault in a component or isolate the faulty component (if it cannot be recovered) to avoid propagating the error.

- Error Reporting: i.e. to report the error to the user if it cannot be recovered autonomously so that the user can either repair the system or shut it down.

- Fault Masking: i.e. producing an output with redundant resources and then selecting the correct output with voting [Gre08]. This technique is rarely used in embedded real-time systems and is not used in SpiNNaker either.

8.3 SpiNNaker Fault-tolerance Support

As explained before, the SpiNNaker computing system simulating biologically-inspired spiking neural networks is a real-time system. The system has *soft* real-time bounds i.e. the crossing of a millisecond limit while updating the neural state of all (1000) simulated neurons in a processor is desired, however, it does not cause a catastrophic system failure. The fact that the nervous system's spike communication is non-deterministic and noisy [Me98, DA01] does not invalidate the real-time behaviour of a simulated neural network. Moreover, the difference

of electronic transmission speed (in the SpiNNaker Communication Network) and the action potential propagation speed (in the nervous system) gives a tolerance of more than a millisecond between the receipt of a spike to its effect as explained in Section 4.5, which enables the delay in missing the update-interval time limit to be caught up.

SpiNNaker uses both error-recovery techniques, i.e. replacement and reconfiguration, to maintain its run-time functionality. The fault recovery process is autonomous and runs as an event-driven application (Section4.5) as part of the SpiNNaker configuration process. The SpiNNaker configuration code for the processors, and particularly for the monitor processor, contains fault-recovery routines to handle certain envisaged faults. The monitor processor in each chip is responsible for dealing with chip-level fault-handing and contributes to the system-wide fault-tolerance with the help of the Host PC, while the other processors continue to run the application. The configuration process uses SpiNNaker's specially-designed fault-tolerance features to provide a support for run-time local recovery, isolation of faulty components, and local reconfiguration controlled by the monitor processor. The following hardware support is provided in the SpiNNaker computing system to support fault-tolerance:

- Fault-detection: Most SpiNNaker CMP components such as the Router, Ethernet Interface, PHY, DMA Controller, Communication Controller, and Watchdog Timer, have error-detection hardware, and report exceptions to the processors as interrupts. The Router and Communication Controller detect and report faults in the Communication Network. They can detect and report on packets received with parity errors, time-phase errors, framing errors, and invalid packet type errors. Besides this, the router reports to the monitor processor on emergency routed packets and packets removed from the network due to congestion on the links to the other neighbouring chips. The router can keep a record of dropped packets while preserving the packet contents. The monitor processor can, if necessary, regenerate the dropped packets to send these via other lightly loaded routes. The Communication Controller interrupts its local processor to handle the error locally, while the router report the error to the monitor processor to handle the fault locally at the CMP level.

- Redundant Processing Nodes: Each CMP has about 20 processing nodes

which are similar in all respects. Any processor can be assigned to perform any specific task as the processing node job assignment has been kept soft and is configured dynamically at run-time. Even the selection of the monitor processor in each chip is kept under software control to support fault-tolerance, as a hard-wired monitor processor introduces a single-point-of-failure. Though the processing nodes' IDs are hard-wired, these are only for configuration purposes. The software can configure their soft addresses to run the same application irrespective of a particular processing node failing. The SpiNNaker application model proposes keeping a few processors as spare to allow run-time replacement.

- Redundant Ethernet Connections: While only one Ethernet connection is necessary to connect SpiNNaker to the Host (and there is a little performance gain from using more than one link during a flood-fill process in a large scale system (Chapter 7, Section 7.3)), we connect the SpiNNaker system to the Host PC using two or more Ethernet connections for redundancy and thus better fault-tolerance. This, again, minimises single points-of-failure.

- Reconfigurable Chip Addresses: The SpiNNaker CMPs are not hard-wired with fixed addresses. The chips are assigned virtual addresses for packet routing at run-time. With this scheme we can implement any networking topology and isolate any non-functional chip(s). This also allows reassigning the job of a dead-chip to a spare one by reconfiguring the Communication Network at run-time.

- Reconfigurable Routing: The router in each chip is fully reconfigurable. Initially, the router can route only NN packets which go to the six nearest neighbours without any knowledge of the chip addresses. For multicast (MC) and point-to-point (P2P) packets, we need to configure the routers to establish links between the processing nodes located in the various CMPs. SpiNNaker allows reconfiguring these routing tables at run-time. This soft configuration and ability to reconfigure at run-time improves system reliability as a broken or congested link can be avoided by local reconfiguration of the routing tables by the monitor processor or externally by the Host PC. Each router contains redundant entries in the routing tables to be reused for reconfiguration. If a part of a routing table is damaged, the remaining

entries can be used to configure routes. A dead processor in a chip or a dead chip in the system can be isolated by configuring the routing tables not to send packets to them.

- Lookup Point-to-point Routing: The chips are arranged in a 2D network with their addresses assigned in 2D Cartesian space along (x,y) axes. The P2P packets are routed logically based on the destination chip's address (x,y) using algorithmic routing. However, we use look-up routing tables for P2P routing to allow tailored routing and reconfiguration in favour of better reliability. The P2P routing table not only helps in isolating a dead chip but can also be reconfigured at run-time to avoid congested routes or to provide multiple paths.

- Emergency Routing: The router can automatically handle a transient fault at its transmission links by using a hard-wired emergency routing mechanism. This mechanism copes with transient faults caused by congestion or other errors on the inter-chip links without having to reconfigure the routing tables. If, however, the fault persists on any particular link or two consecutive ports are blocked, causing packets to be dropped due to failed emergency routing, the router informs the the monitor processor after dumping the dropped packet into its registers. The monitor processor can reconfigure the routing table to avoid sending packets to this link, after analysing the situation at the router.

- Watchdog Timer: Each SpiNNaker CMP contains a watchdog timer to trap a non-responsive chip. The purpose is to monitor the monitor processor's health in each chip and to contribute to the system-wide management. The monitor processor needs to reset the watchdog timer after a specified interval, failure to do so resets the monitor processor or the whole chip.

- RAM-ROM Remapping: Each chip boots from the Boot ROM on power-on. We keep the minimum possible code in the Boot ROM to support only initial testing and initialization of the chip resources. However, to avoid a single point-of-failure in the case of a malfunction in the Boot ROM, the SpiNNaker CMP can use the System RAM in place of the Boot ROM. The System Controller can remap the Boot ROM address in the System NoC to point to the System RAM.

- System Controller: The System Controller on each SpiNNaker CMP plays a vital role in chip management and local fault-tolerance. It maintains the state of all the processors and chip resources which can be viewed by all the neighbouring chips using the NN peak-and-poke feature. The System Controller has been designed to allow a reset of only selected processing cores, a selected processing node (a processing core along with its supporting peripherals), selected chip-components (sub-systems) in the chip, or the whole CMP. The purpose is to recover locally from deadlocked states. Besides this, the System Controller also contains support to disable any processing node in order to isolate a faulty processor. The inter-CMP transmission and receive links can also be reset or disabled with the help of the System Controller.

8.4 Fault-Tolerance in the Configuration Process

The SpiNNaker configuration process makes use of the fault-tolerance features in the SpiNNaker design to provide a fault-tolerant hardware platform to the user. The following sections describe some important fault-tolerance features of the SpiNNaker configuration process.

8.4.1 Monitor Processor Selection

Contrary to a typical CMP-system where a monitor processor is hard-wired at design time, thus introducing a single point-of-failure, SpiNNaker chips do not have a dedicated monitor processor. The configuration process selects the monitor processor from the healthy processors for better fault-tolerance. The monitor process can be replaced by any other healthy processor at run-time, in case of any problem with the monitor processor.

8.4.2 Boot ROM Failure

In case of a Boot ROM failure, the configuration process makes use of the RAM-ROM address remapping feature to use System RAM for the boot-up process with the help of a neighbouring chip. The process is described later.

8.4.3 Chip-level Recovery

To detect and recover a faulty SpiNNaker CMP a detailed chip-level recovery mechanism has been devised which uses specially-designed features in the chip. During chip-level initialization, the monitor processor tests all the devices and tries to reset any faulty components. This may allow a device to recover from a transient fault. If, however, a device does not respond after reset, the monitor processor records its state and informs the Host PC while reporting the chip's state. Similarly, each processing node informs the System Controller of its state which is recorded in a special register. The monitor processor tries to bring a dead processing node up by resetting it, or informs the Host PC. During application execution, all the processors' activity (sleeping or active) is recorded in the System Controller. The monitor processor can poll this information to examine any non-responsive processor. A suspected processor can be sent an interrupt to see if it responds, a failure to do so may warrant the monitor processor resetting the faulty processing node.

The chip-level management and recovery depends on the health of monitor processor, which itself may be deadlocked due to a hardware or software fault. A Watchdog timer in each chip detects a non-responsive monitor processor and informs the System Controller to reset it or replace it with a healthy one. The Watchdog generates an interrupt after a specified interval and a reset signal if the interrupt is not cleared in the next interval of the same length of time. The Watchdog interrupt line is connected to all the processing nodes, besides being available to the System Controller. The System Controller contains a special register to record the reason for the last reset i.e. the power-on reset, Watchdog chip-level reset, or a Watchdog soft reset to only the monitor processor. The System Controller allows setting an option to reset the monitor processor on the Watchdog interrupt for configuration flexibility. We implemented the following two options in our configuration process to recover a non-responsive chip:

- Reset on Watchdog Interrupt: with the help of the System Controller, we can configure the monitor processor's soft reset on receiving the Watchdog Timer's interrupt. If this feature is enabled, the monitor processor is required to reset the Watchdog's counter periodically before the Watchdog generates an interrupt. Failure to do this will cause the System Controller to generate a soft reset to the monitor processor i.e. only the monitor processor (without its peripherals) will be reset. It is an efficient process which

relieves the monitor processor from handling an interrupt while doing its processing. Resetting the watchdog timer is only a 2-3 instruction routine as compared to the interrupt service routine which takes 12-13 instructions besides saving the context and changing the operational mode during which the monitor processor may have to divert from an important job.

- Event-Driven Reset: The previous option is efficient. However, it requires the monitor processor to continuously take care of resetting the watchdog before the lapse of the Watchdog's configured interval. Contrary to this, the second option uses an event-driven chip-recovery process whereby the monitor processor enables the Watchdog interrupt in its interrupt controller and clears the interrupt whenever it is received. Failure to do this causes the Watchdog to generate a reset signal to all the processing nodes, eventually resetting the whole chip. This feature is dangerous in that the application processors and the chip will lose their state as a result of only the monitor processor's misbehaviour.

The two options are available to the user and the configuration depends on the user's preference.

8.4.4 NN Diagnostics and Recovery

As described in Chapter 4, the MCRouter on each chip can support MC, P2P and NN packets. An NN packet can travel only between two adjacent chips. However, with the help of an NN packet a chip can broadcast a packet to all its 6 neighbours at the same time, can communicate with any of its chip individually, or peek-and-poke any of its neighbouring chip's resources. The peek and poke feature of the NN packet has been designed specifically to help bring up a dead chip during the configuration process. Using this feature, the chips surrounding a dead chip try to diagnose the reason by reading the information recorded in the dead chip's System Controller.

As described in Chapter 7, at the end of the configuration process phase I, each chip tries to check its connections with six other chips by reading the neighbouring chips' status from their System Controllers using NN packets. If a chip does not get a response packet back or the status indicates a problem in any chip, the six chips neighbouring a dead chip activate a "neighbour diagnostic" routine to recover the dead chip.

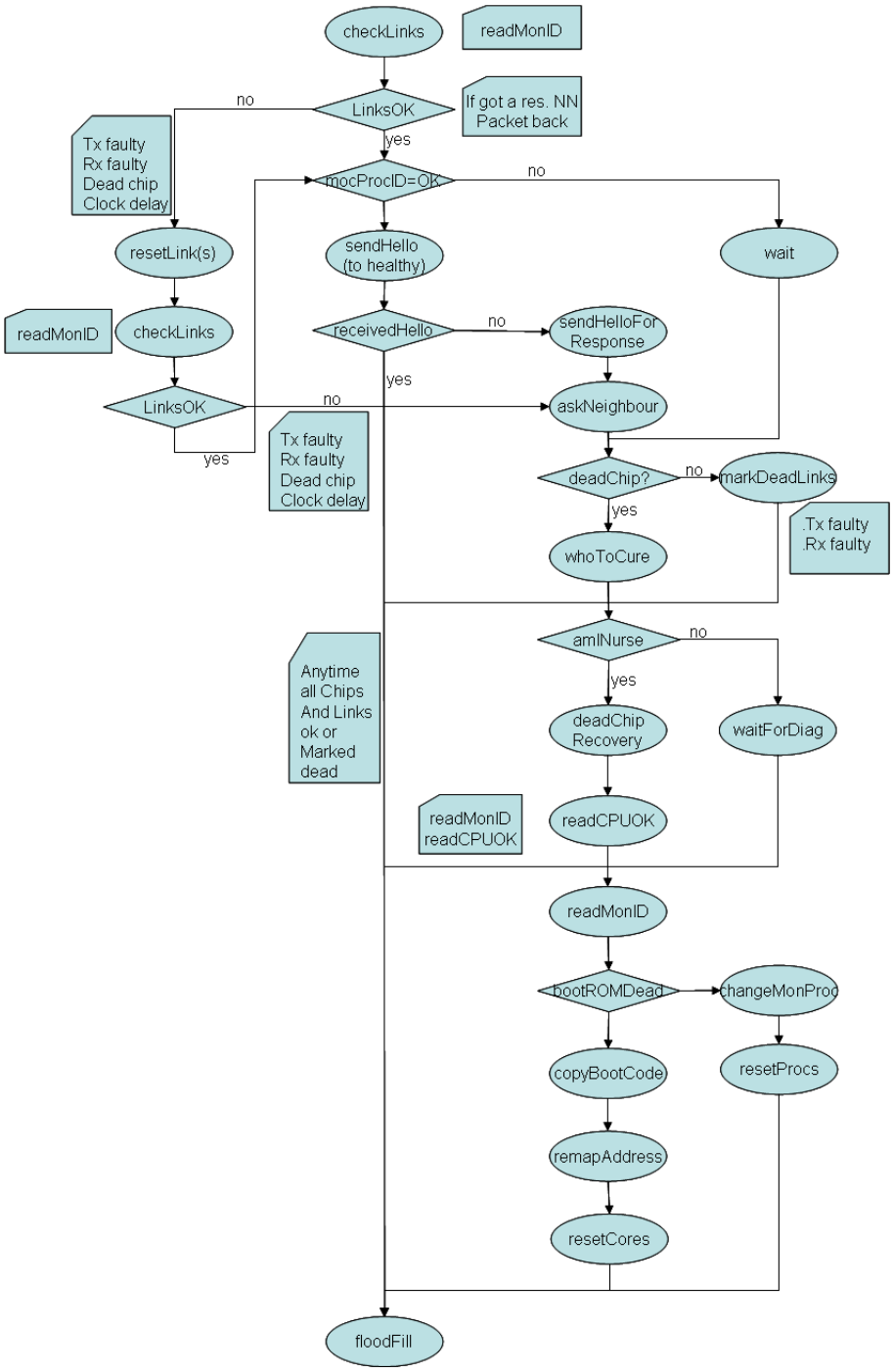

Figure 8.1: NN Diagnostic and Recovery Process.

The following process of neighbour diagnosis and recovery, as graphically represented in Figure 8.1, is adopted by the monitor processor on all the chips:

- If a chip fails to read from a link after a specified interval, there is a possibility of a dead chip or broken links between the two chips. The chip on either side resets its non-responsive links before checking the link again. The process is repeated a number of times to ensure recovery from a transient error.

- If a link does not respond after a number of resets, the chip marks the link as dead.

- Each chip sends a Hello message to the chips with live links.

- If a Hello message is not received during a specified interval or the chip status, read earlier while checking the links, indicates a problem in the chip, the NN Recovery process in the neighbouring chips tries to recover the dead chip. For this purpose, a "nurse chip" is selected from the healthy neighbouring chips with live links to the dead chip. The process of nurse chip selection will be described later.

- The nurse chip reads the monitor processor ID register and the processor's status registers from the dead chip's System Controller to diagnose the problem. If the monitor processor has been selected and the chip resource state indicates more than one healthy processor, the nurse chip changes the monitor processor ID register to point to another healthy processor, disables the dead monitor processor, and resets the newly selected processor to take over as the monitor processor. This will avoid reinitializing the whole chip and any disruption to the application running on the application processors, and keeps the state of the chip intact. The nurse chip can also reset all the processors in the initial configuration process to restart the selection of the monitor processor. In this case, the System Controller will ensure that the same processor is not selected as the monitor processor.

- If the System Controller is unchanged, the nurse chip tries to read the Boot ROM and does some basic testing to examine its state. If it discovers any problem with the Boot ROM, the nurse chip copies the code from its own Boot ROM into the System RAM of the dead chip, and then remaps

the address of the Boot ROM in the System NoC to the System RAM. It then resets all the processors to restart the chip-level boot process from the remapped location.

- If the nurse chip can not determine a viable recovery solution, it resets the whole chip in an attempt to recover from an unknown transient problem.
- if nothing works, it isolates the chip by disabling the clocks of all the processors on the dead chip and reports the matter to the Host PC.

Selecting a nurse chip from the six neighbouring chips surrounding a dead chip is an interesting issue in a fully distributed system. We considered the following three options for selecting the nurse chip:

- Fixed: a fixed neighbour is always selected as the nurse chip, e.g. the chip on link '0' of the dead chip may be selected to perform this task. This is the easiest implementation for the dead-chip recovery process. However, it introduces single point-of-failure i.e. the process cannot proceed if the link or the chip on port '0' of the dead chip is out of order.
- Central: the chips report their state along with the state on their links to the Host PC as part of the configuration process phase II. After acquiring the state of the whole system, the Host PC nominates one of the dead-chip's healthy neighbouring CMPs with active links to cure the dead chip. The protocol is simple and does not introduce single point-of-failure, however, the process is not autonomous and will take a while to cure a dead chip as the Host PC will nominate a nurse chip only after receiving the state of all the chips.
- Dynamic: the six neighbouring chips collaborate to decide which one of them should become the nurse chip. For this, the six neighbouring chips read the arbitration register of the dead chip's System Controller used to select the monitor processor, and the one getting the first access (a return value 0x80000000) is selected as the nurse chip (this uses the monitor processor selection process as explained in Chapter 7).

8.4.5 Connection to the Host PC

An Ethernet interface has been provided on each SpiNNaker CMP to connect it with the outside world as described in Chapter 4 (Figure 4.10). Using the on-chip Ethernet, a single SpiNNaker CMP can be configured to work as a fully functional SpiNNaker system connected to the Host PC, which can simulate a neural network of up to 20,000 neurons. The process can be refined to allow a multi-CMP SpiNNaker system to be divided into more than one subsystem to run different neural simulations at the same time by configuring its routing tables to partition the system and connecting each partition to the Host PC using Ethernet connections. In an integrated multi-CMP SpiNNaker, we can configure more than one Ethernet link to connect the system to the Host PC to provide redundant links. The results in Chapter 7 show that loading the application with more than one link does not improve the performance of the process. However, the redundant links improve the reliability of the process by minimising single points-of-failure as a single link can malfunction at run-time to lose the connection with the system.

8.5 Fault-Tolerance in Application Loading

The application load process uses broadcast (or multicast) to flood-fill the application and its data into the chips. As a consequence of the flood-fill process, each chip receives redundant data, ensuring delivery of all the data and its associated application. Chapter 7 shows that the 2Msg forward technique to flood-fill data is the best in performance, however, this does not ensure delivery of all packets to each chip twice. The configuration process favours an approach that delivers redundant packets to each chip for fault-tolerance. In the case of no redundancy, a chip expecting to receive packets only from a broken link will remain void of data and, thus, will not pass the data onward, leaving all the chips in that direction without the application. The redundant packets can also be used to verify the data by comparing the two packets. While each chip thus receives the same application, the monitor processor configures the application according to its chip location. Since each chip contains the the same application, any chip can be reconfigured at run-time to take on the role of some other chip, and can likewise locally provide missing data to a chip that recovers from its initial faulty state.

8.6 Evaluation Work

8.6.1 Chip-level Recovery

We experimented with the chip-level recovery process in the case of the monitor processor's malfunction at run-time. The Watchdog timer was configured to detect a non-responsive monitor processor and notify the System Controller. The following scenarios, as explained in Section 8.4.3, were implemented and tested:

- Reset on Interrupt: For this test, we configured the Watchdog timer to generate an interrupt after every 20ms. The System Controller was configured to generate a reset signal to the monitor processor on receipt of an interrupt from the Watchdog timer. We configured the Timer Controller interrupt service routine for the monitor processor to reset the Watchdog timer's counter after every 9ms. This ensures that the watchdog counter is reset twice before a reset could be generated. We then disabled the Timer interrupt to let the watchdog generate an interrupt after 20ms. The monitor processor was reset by the System Controller (using a soft-reset i.e. only the monitor processor itself, and not its peripherals). The boot-up process checked the reason for the reset code (which was the Watchdog Interrupt as expected) in the System Controller before starting the chip-level initialization and avoided the testing and initialization of the chip resources as the state in the System Controller indicated healthy chip resources. The process took ∼0.9ms to bring the chip up as compared to the normal ∼1.3ms from a cold start.

- Event-Driven Reset: In this option, we configured the Watchdog to generate an interrupt after every 10ms and then to generate a reset signal after the same interval. The monitor processor was configured to enable the Watchdog interrupt in its interrupt controller, and cleared the interrupt in the Watchdog timer to avoid getting a reset after another 10ms. We then disabled the watchdog interrupt in the monitor processor interrupt controller to allow the Watchdog to generate a reset signal to all the processors. After 20ms all the processors in the SpiNNaker CMP were reset by the Watchdog timer. The process took the normal chip-level cold boot-up time (∼1.3ms) as the whole chip was configured again .

8.6.2 NN Diagnostics and Recovery

Here we present our results for the NN diagnostics and recovery process. We evaluated the following scenarios with this process:

- Inter-CMP Broken Links: The links between Chip(0,0) and Chip(1,0) in a 4-chip SpiNNaker system were disabled to test this situation. In the first case, we temporarily disabled the links in the SpiNNaker SystemC simulation in a way to be fixed on reset. In the second case, we permanently broke the links between the two chips. In the first case the chips reset the links and recovered from the problem, while in the second they marked their links as broken after enquiring from their common neighbour. The two situations took different times as shown in Table 8.1.

- Monitor Processor Non-responsive: In this case two situations were tested. In the first situation we recovered the dead-chip by replacing a monitor processor, disabling the first one and sending a reset to the newly selected monitor processor to take over. In the second situation, we reset all the processors to restart the boot-up from the scratch. Table 8.1 shows the time taken by the two recovery processes.

- Boot-ROM Broken: In this case, the boot-up code was not copied to Chip(0,0) in a 4-chip SystemC simulation of a multi-CMP SpiNNaker. We implemented the fixed technique for selecting Chip(1,0) as the nurse chip to bring the dead chip up. Table 8.1 shows the time taken by this process to bring up the dead chip (Chip(0,0)).

Table 8.1: NN Diagnostic and Recovery Process.

S/No	**Fault**	**Measure Taken**	**Time**
1	Inter-chip link broken	Each Chip resets its non-responsive links. The process is repeated a number of times after a delay of 3ms before giving up.	0.48μs.

2	Inter-chip link broken and could not be repaired	Disable link and communicate through a common neighbour. The non-responsive link was reset thrice at 3.9ms, 6.1ms and 8.9ms after the boot-up. After three resets, the link was disabled and marked as dead in the ITCM to avoid sending packets to this port.	6ms
3	Monitor processor not responding	Option 1: (if the chip configuration has already been done by the previous monitor processor) the nurse chip changes the monitor processor ID to another processor, disables the current monitor processor and resets the newly selected processor to take over as the monitor processor	1.09ms
4	Monitor Process not responding	Option 2: (if the chip has not been initialized) the nurse chip resets all processors to restart the boot process.	1.37ms
5	Boot ROM broken	The nurse chip loads the Boot ROM image to the dead-chip's System RAM, remaps RAM-ROM address and resets processors to boot from the System RAM.	13.2ms (12ms to load the boot ROM)
6	Nurse-chip failed to bring the dead-chip up	disable all processors' clocks in the dead chip and report to the Host PC	1.2μs.

8.6.3 Application Loading

We have conducted various experiments to prove that the application loading process does succeed in various situations of faults in the Communication Network. We considered the following link failure modes:

- Vertical Compartments: all the links along the x-axis in a 2D torus configuration of the SpiNNaker multi-cmp system were treated as blocked, leading to a network split into multiple columns. We did not, however, disable the diagonal links, so the network is not completely split.

- Horizontal Compartments: all the links along the y-axis were disabled, to partition the network in multiple rows in a SpiNNaker 2D torus configuration. As above, the diagonal links remained intact.

- Cross Links: the union of the previous two, i.e. both the horizontal and vertical links were disabled splitting the network into four square sub-networks. The chips could only communicate through their diagonal links.

- Random Link Failure: a random set of link failures was tested in this case. We tested systems with various degrees of network degradation. We tested the system with random link failures of $L/256$(rnd1), $L/128$(rnd2), $L/64$(rnd3) and $L/32$(rnd4) links, where L is the number of total links (64K*6=384K) in the system. It gave 1K to 64K link failures.

We tested the system using 1, 2 and 4 Ethernet connections to the Host PC from the CMPs located at (0,0), ($X/2$,$Y/2$),($X/2$,0) and (0,$Y/2$) where X and Y are the number of CMPs along X- and Y-axis respectively in the SpiNNaker 2D toroidal configuration (Chapter 4 Fig. 4.10). Finally, we tested different network sizes, all of them square, ranging from 32x32 to 256x256. Figure 8.2 shows the effect of the number of Ethernet connections on the application loading process in the presence of faulty inter-CMP links as explained above. Though connecting the Host PC at more than one point to the SpiNNaker system does not improve the application loading time in a large-scale system (Chapter 7 Section 7.5), it does improve the fault-tolerance of the process as a chip can receive a packet from various directions. It is particularly important if the network is split in various regions due to link failures. Here, the broadcast mechanism proves to be the most robust by losing no packets in any failure setting, while the 3Msg forward

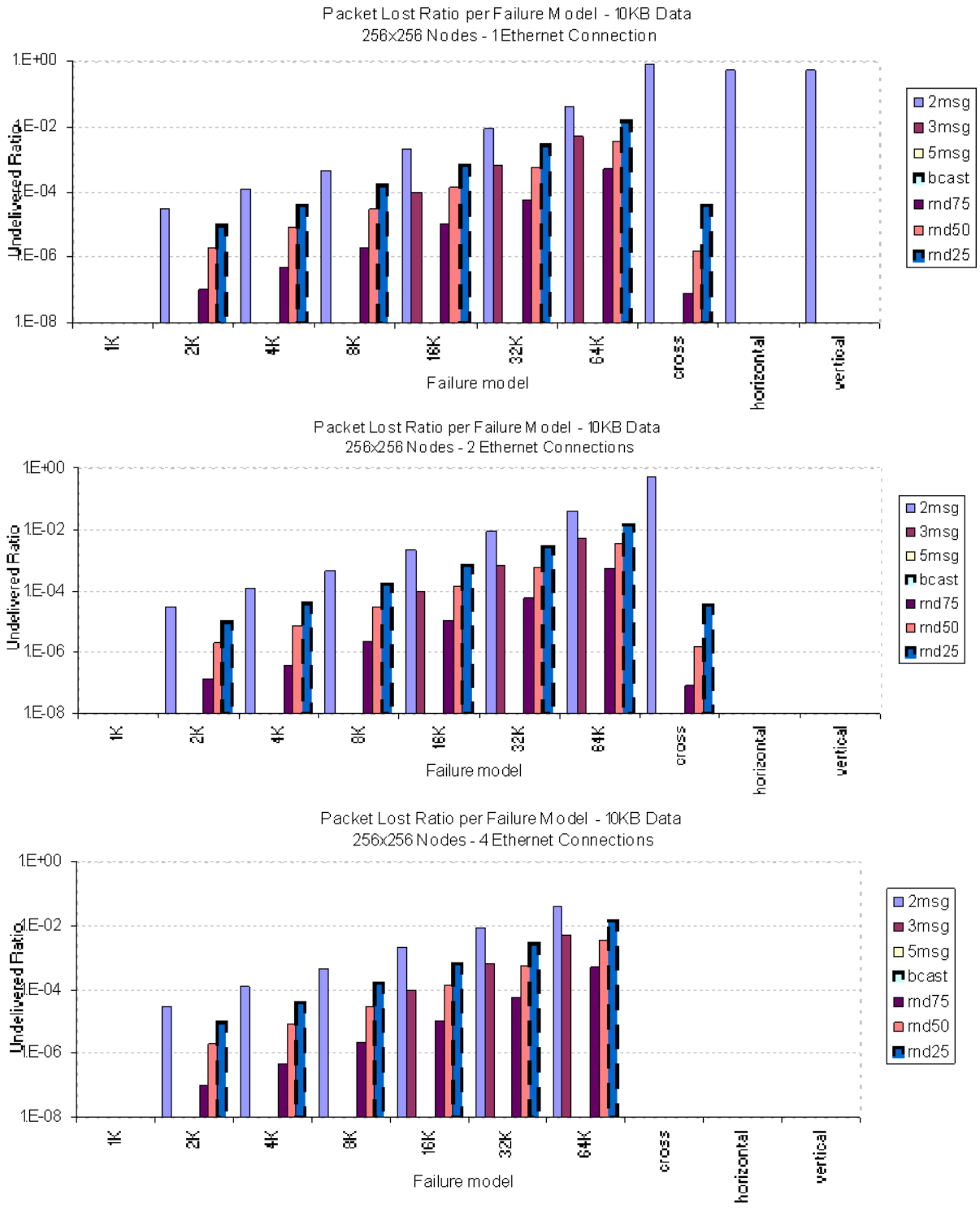

Figure 8.2: Fault-tolerance in Application Loading Process with Varying Numbers of Ethernet Connections [KNR+09]

provides reasonably good fault-tolerance as it is not affected by horizontal and vertical failed links and random link failures up to 8K (a system degradation of about 2%). The 2Msg forward technique is the worst in robustness, though the best in performance as proved in Chapter 7.

8.7 Summary

Fault-tolerance in hardware systems is not a new idea. Many clever techniques have been devised to make computing devices sufficiently reliable, especially mission-critical embedded systems. Nature is known for engineering reliable systems, and our brain is one such example. The history of computing system failures suggests that it is worth spending more time and resources in order to improve the fault-tolerance of our products. May be we can learn from nature how to do this better. In the SpiNNaker project, one of our objectives is to design a high-performance and fault-tolerant computing system in order to explore nature's engineering methodologies in producing fault-tolerant systems. We use known fault-tolerance techniques from computing systems and provide support for experimenting with the techniques biology uses to make its systems fault-resilient. The SpiNNaker computing system provides support to detect a number of faults and recover from these with the help of software. SpiNNaker has been designed as a modular and reconfigurable system with redundant resources and specially-designed fault-tolerance features to provide a reliable real-time hardware neural simulation engine. The SpiNNaker configuration process makes use of these features to bring the system up reliably and to load applications into the system. SpiNNaker running a spiking neural network in biological real time acts like a real-time system. As a real-time system has to be fault-tolerant to perform in real-time, the SpiNNaker configuration process ensures this in the absence of an operating system. The experimental results shows the performance of some of the techniques implemented as part of the SpiNNaker configuration process.

Chapter 9

SpiNNaker Hardware Abstraction Layer

9.1 Introduction

The SpiNNaker computing system is an Application Specific Integrated Circuit (ASIC) design composed of off-the-shelf Intellectual Property (IP) hardware resources from various vendors and in-house designed components to perform specific functions in the context of the overall SoC design objectives. The target users are multi-disciplinary scientists and engineers developing or using neural network applications in the neural computation research space. Present day neural applications are being developed using a variety of software tools targeted at conventional computing systems. The main motivation for the SpiNNaker design emerges from the limitations of general-purpose computers in terms of their performance when simulating large-scale neural networks using biologically-inspired neural models (as described in Chapter 3). There is a large collection of developed applications based on decades of research work ready for a suitable hardware platform. The SpiNNaker multi-CMP massively-parallel computing system is one realization of a long-awaited need to enable the simulation of large neural populations in their biological real time. With a large-scale neural network simulator, such as SpiNNaker, there is now a need to port these applications to this system without undue concern with the architectural details of this computing device. The solution is to devise an abstraction layer to interact with the hardware and provide a high-level Application Programming Interface (API) to application developers without the knowledge of the underlying hardware and to enable them to

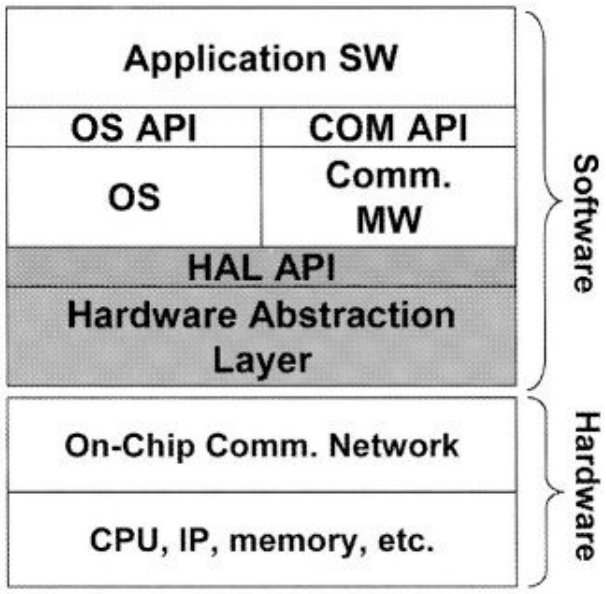

Figure 9.1: Hardware Abstraction Layer [SY03].

develop their own applications without detailed knowledge of the physical-layer complexities.

This chapter highlights the importance of developing a hardware abstraction layer in a computing system, especially in a purpose-built computing device that is architecturally unique from a normal software developer's perspective. The chapter describes the contribution as part of this research to provide an abstraction layer in the form of a library of useful functions to facilitate the development of neural applications without needing much knowledge of the SpiNNaker architecture. In conclusion, some evaluation work performed to test the correctness and performance of some of the useful functions is presented.

9.2 Hardware Abstraction Layer

Software component reuse necessitates a hardware dependent software API at the lowest level to interact directly with the hardware and to provide a common interface to the applications running on top. It is used to port the operating systems and applications on varying number of computer architectures easily. The interface is called the "Hardware Abstraction Layer" (HAL) or the "Board Support

Package" (BSP) [SY03]. The HAL comprises all the software "directly dependent on the underlying hardware" [SY03] and may constitute boot-up instructions, code for configuring and accessing hardware components (device drivers), context saving and switching mechanisms, interrupt service routines to handle device-level interrupts during application execution, and device-exception handling routines etc. Figure 9.1 shows how a HAL should provide an interface between the hardware and the application. In its simplest form a HAL can be considered as a "nano-kernel" which is a combination of interrupt service routines and task stacks [PBK91]. Mostly HALs form part of an operating system and their definition is also specific to the vendor's operating system. For example, in the case of Windows CE, the HAL or BSP consists of boot-up instructions, the OEM abstraction layer, device drivers, and configuration code [SY03]. A variety of device drivers are provided in a library, from which ones specific to the architecture are installed with the help of a configuration tool called "Platform Builder" [SY03]. Other commercial operating systems follow similar techniques.

For SoC design, especially in CMP architectures, besides software reuse, the HAL has to deal with concurrent processes/threads which requires a notion of synchronization using some mechanism of inter-process communication. In the SoC design cycle, using a system-level modelling approach for hardware/software co-design, we can finalize the hardware/software interface early in the design process to start developing the software in parallel with the hardware design. Thus the HAL can be developed along with the application. In such a case the application can be developed on a PC without reference to the hardware architecture which would be taken care of by the HAL. The Virtual Socket Interface Alliance (VSIA) [All00] has been trying to standardize its specifications for developing a HAL API to make it acceptable to most hardware and software vendors. However, in the case of SoC designs having application-specific architectures with newly designed components, a fixed standard API may not be feasible. Sungjoo et al. [SY03] highlight the following three major issues while implementing a HAL for an SoC design.

- HAL Modelling: In the case of an SoC the software development starts in parallel with the hardware design, which initiates the specifying and implementing of a HAL for the hardware which is still in the design phase. Because the target hardware is not ready, the HAL cannot be verified. This requires modelling of the HAL along with modelling the SoC during

the initial phases of hardware/software co-design. The modelled HAL can then be tested on the TLM or RTL system-level model.

- Application Specific HAL: A generic HAL API may be suitable for most available hardware components, however, if an SoC has been designed for a specific application with many in-house components, it may be easier to devise a new HAL than to port a nano-kernel to conform to the newly devised architecture; this is the case with the SpiNNaker architecture. A case study conducted to find a suitable off-the-shelf micro-kernel for SpiNNaker concluded that it would be easier to write dedicated device drivers and other functional routines than to use an existing open-source embedded microkernel.

- Manual HAL Design: For an SoC design, it is necessary either to implement the HAL manually or to configure it for a specific set of components on the SoC board. Even with a configuration tool, it is necessary to extend the API by writing drivers for components not supported by the library. The process is time consuming and error prone for SoC designs.

9.3 Abstracting SpiNNaker

The SpiNNaker HAL has been implemented manually because of its specific-to-application design and the use of in-house designed components, especially in dealing with inter- and intra-CMP communication. Most of these functions have been implemented in ARM assembly-code for optimal performance to support real-time applications. The SpiNNaker HAL consists of the following classes of functions.

9.3.1 Boot-up Instructions

The SpiNNaker HAL includes the implementation of the boot-up process described earlier in Chapter 7. The code includes instructions for the reset handler, stacks and heap initialization, the component-level Power-on-Self-Test (POST), local and chip-level memory tests, processing node and chip-level device test, and the initialization of these devices for default functionality. The instructions also load the boot-up code into the local instruction TCM of each processing core. As described in Chapter 7, in the boot-up process the instructions configure the

SpiNNaker CMP in phase I and helps with system-level integration in phase II of the configuration process. The instructions also configure the chips to support application loading into the system from the Host PC.

9.3.2 Neural Support Functions

The SpiNNaker HAL API provides certain routines to facilitate neural simulation on SpiNNaker. One such facility is the support of inter-neuron spike communication. The HAL provides functions to the programmer to facilitate sending and receiving spikes. The functions communicate with the Communication Controller to emulate sending spikes using Multicast packets determined by the parameters provided by the programmer. The library also provides functions to set up routing tables in the on-chip router to facilitate inter-neuron communication. This function also ensures that the router is configured to disable unused entries in the routing tables to discard any unwanted packets from the network. The HAL library provides functions to support the synaptic data transfer using DMA operations to provide a localised view of the synaptic data. Again care has been taken to make the functions optimal by using design features of the DMA Controller, System NoC and the SDRAM controller (e.g. performing double-word transfers with maximum burst size to make full use of the AXI interface of the System NoC to access SDRAM). Similarly the functions report to the application if an already-requested DMA operation is in process.

9.3.3 Interrupt Service Routines

The HAL library implements the Interrupt Service Routines (ISR), including the context saving and restoring mechanisms, for all possible interrupts. As the SpiNNaker application model (Chapter 4) uses these ISRs to execute the spiking neural application as an event-driven real-time application, an entry point is provided in each ISR to the user's application. A list of these entry points will be made available to users for application development. Users can employ their own implementation of these functions while developing applications on a PC, however, "including" the SpiNNaker HAL library ensures that the user defined functions are "called" in the ISRs to integrate the application with the configuration process code.

9.3.4 Optimal and Safe Device Interface

The SpiNNaker HAL is intended to make an optimal and safe use of the SpiNNaker-CMP resources. Accessing the hardware using the device driver functions will ensure that application-specific features of a device, unknown to a programmer writing a high-level application, are made use of. This also ensures the avoidance of any undesirable access to a feature kept for diagnostic purposes, or an accidental misuse of a feature. For example, the optimal DMA operation is with double-word size and a burst size of 16, however, a double-word access to the System RAM is not supported and generates an AXI error in the DMA. Similarly, burst size has no effect on the Router access through DMA operation as the Router is provided with an AHB interface. The HAL functions for DMA operations ensure that a function with appropriate options is initiated based on the target devices. The Communication Controller has been optimally designed to support neural spike communication i.e. a spike can be sent by just one ARM instruction to write in one of the Communication Controller's register provided the configuration process has configured the Communication Controller properly. The library provides the default "sendSpike()" function along with a few more functions with various options to support interprocess communication using multicast (MC), point-to-point (P2P), and nearest-neighbour (NN) packets. Sometimes, sending a packet may require writing the control byte and the source-chip address to the control register (R0) of the Communication Controller, which contains some "sticky" information bits indicating the transmission side buffer status to help the application. An accidental clearing of these bits may provide false information to the application resulting in packets being lost due to congestion in the Communication Network. The sendPacket() function from the SpiNNaker application library takes care of these aspects.

For proper functioning of the Router, the Router's initialization routine in the SpiNNaker HAL sets the unused entries in the Router's Mask table to $0x00000000$ and those of its Key table to $0xFFFFFFFF$, which invalidates every bit in the packet routing key ensuring that a packet with an invalid routing key is rejected and removed from the Communication Network [Pro07]. Similarly, the unused entries in the Router's point-to-point table are initialized to be directed toward the local monitor processor and removed from the network, or a P2P packet with a certain invalid destination key in a point-to-point packet may keep roaming the SpiNNaker Communication Network causing undesirable congestion. The router

initialization routine makes sure that the MCRouter is configured properly.

Chip-level shared variables are maintained in the shared memory to maintain chip-level configuration status. Similarly, for inter-processor message passing, the processors' mailboxes are kept in the shared memory. In the case of RAM-ROM mapping, a neighbouring chip can load its Boot ROM code into the dead chip's System RAM to be used as the Boot ROM, in which case the System RAM can not be used for this purpose. There is no hardware memory protection unit in the SpiNNaker CMP, so it is the responsibility of the programmer to take care of these memory locations. The SpiNNaker HAL takes care of these addresses in its functions to protect the shared memory locations.

9.3.5 Device Exception Handling

Most devices on a SpiNNaker CMP generate exceptions on detection of a fault, as explained in Chapter 8. These faults are reported as interrupts to all the processors, however, the interrupts from the chip-level shared devices are serviced by the monitor processor. The interrupt handlers for these interrupts call various error handling routines from the SpiNNaker HAL library to diagnose the errors and provide a remedial measure where possible. The application can direct these handlers to custom error handling codes by changing the entry points in the ISRs.

9.3.6 Fault-Recovery Procedures

Chip- and system-level fault-handling routines have also been implemented as part of the SpiNNaker HAL library. These provide support for system-wide fault-tolerance mechanisms as explained in Chapter 8. The routines include handling chip-level recovery as a result of chip-level reset by the Watchdog timer, and functions to cure a dead chip as a result of the nearest-neighbour diagnostic process explained in Chapter 8. The SpiNNaker HAL also provides a limited support to manage fault-handling externally from the Host PC at run-time.

9.3.7 Shared Memory Message-passing

Besides providing inter-neuron communication using packets on the SpiNNaker Communication Network, the HAL library also provides a mechanism for shared memory inter-process communication among the processors within a chip. This has been implemented using a locking mechanism in the System Controller to

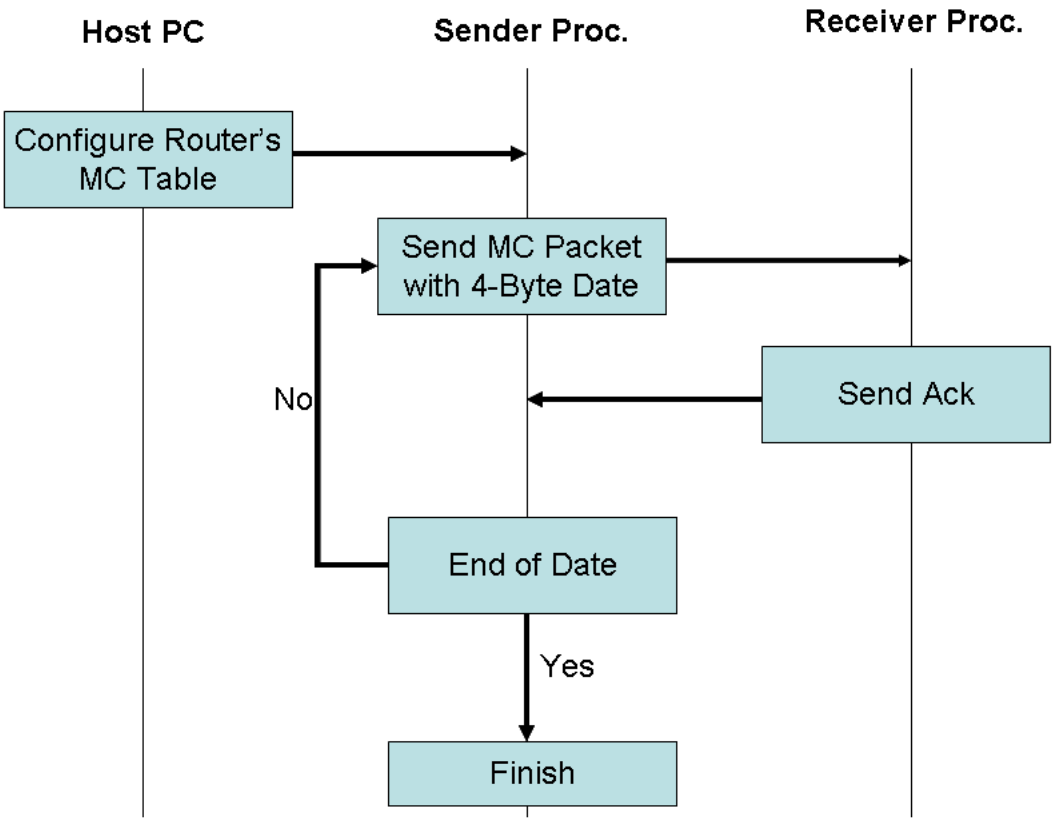

Figure 9.2: Sequence Diagram - SpiNNaker On-chip Interprocessor Message Passing using MC packets.

obtain a lock to the destination processor, individual mailboxes in the shared memory (system RAM or SDRAM) for each on-chip processor to place the data to be sent, and the processor interrupt mechanism in the System Controller to notify the destination processor about the message. This mechanism is used to reduce traffic congestion on the Communication Network, to transfer data among on-chip processors.

We have devised the following three message passing methods with varying performance depending on the data size as shown in Figure 9.4.

- Multicast Packets: This method uses the on-chip router to pass messages as small packets carrying 32-bit data. This process requires configuring the router to route packets to particular on-chip processors, and then send the data with the help of a series of multicast packets. The routing key is used

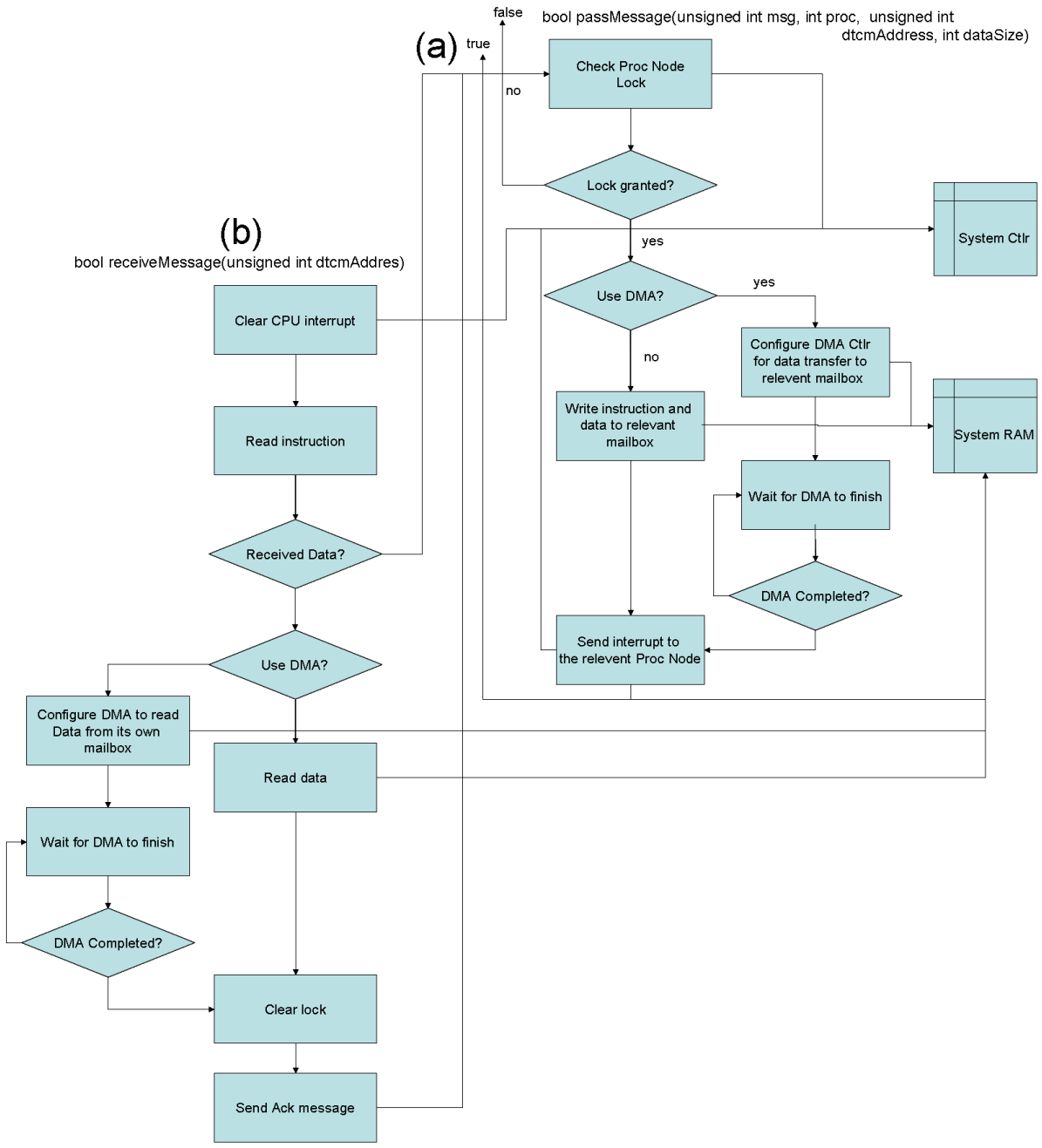

Figure 9.3: Interprocessor Message Passing in a SpiNNaker CMP using Shared Memory, (a) Send side, (b) Receive side.

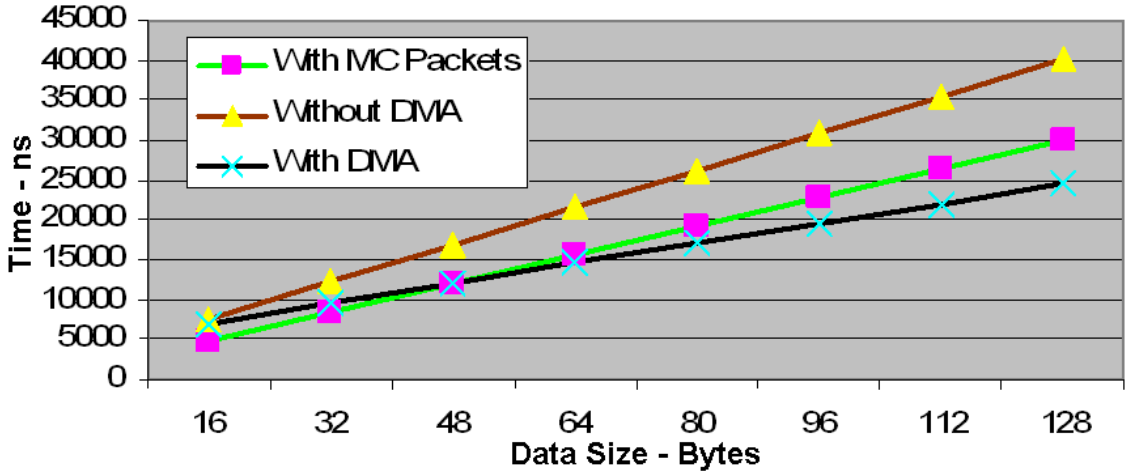

Figure 9.4: Analysis of Interprocessor Message Passing Techniques. [SY03].

to indicate the destination processor along with certain instruction bits to the destination processor about the handling of data. Figure 9.2 shows the message passing algorithm using multicast packets.

- Shared Memory without DMA: This method uses on-chip shared memory (System RAM or SDRAM) to pass messages. The process is shown in Figure 9.3. The DMA operation is not used in this process as sometimes the DMA may be busy doing a more important job such as transferring synaptic data in and out of SDRAM etc. The data write/read operations are performed as single word write/read to/from the mailboxes. The method is more efficient than the one with DMA operation for small sizes of data as it does not involve the overheads involved in configuring the DMA.

- Shared Memory with DMA: This method is the same as above except we use DMA operations to read/write data to/from the shared memory. The method with DMA operations is more efficient for large sizes of data, as it uses the System NoC optimally with the maximum word and burst size allowed by the target device.

.

To ensure uninterrupted communication, the process uses a specially-designed locking (mutual exclusion) mechanism in the System Controller to lock access to a processor in the SpiNNaker CMP that has not already been locked. Figure 9.3

shows the algorithm for the message passing mechanism using shared memory. Each processor has been provided with a private mailbox of variable length (controlled by the application) to enable crossbar inter-processor communication. The first word in the mailbox is used to convey a meaningful message, followed by the data to be transferred. After acquiring a lock, the source processor writes an instruction in the first word of the mailbox and if data needs to be sent to the target processor, this is written in the remaining mailbox. The source processor sends an interrupt to the target processor with the help of the System Controller. The target processor reads the instruction in the first word of its mailbox and reads the data from its mailbox or writes the required data as a new message-passing request. At the end of the message receipt, the target processor clears the lock to itself and sends an acknowledgment interrupt to the source after writing an acknowledgment message in its mailbox.

9.3.8 SpiNNaker-Host Communication

As discussed in Chapter 7, one of the challenges of configuring a multi-CMP system is to provide a mechanism whereby the user or system administrator can communicate with the application and the system. The SpiNNaker HAL library includes functions to support this communication at run-time using Ethernet frames and NN or P2P packets depending on the broadcast or point-to-point message. The library also supports functions for the NN packet communication used for system-level configuration in phase II of the configuration process and the application loading process as explained earlier in Chapter 7. The library also provides support to transform between the two types of communication (i.e. Ethernet frames and NN packets) on the Host-connected chips.

To handle the Ethernet communication, the Ethernet device driver implemented as part of the HAL library provides optimal functions to send and receive frames. The functions provide support for data transfer between the Ethernet controller and the local memory with or without the DMA operation depending on the size of data in order to avoid the DMA configuration overheads in the case of small size frames. We have devised an instruction set to be used as a communication protocol between the Host PC and the Host-connected CMP. A 4-byte instruction is passed along with 2-byte length of data after the TCP/IP headers for this purpose. A few options as part of flood-fill or CMP-specific instructions are also passed in 12-16 bytes after the instruction. The frame format used in this

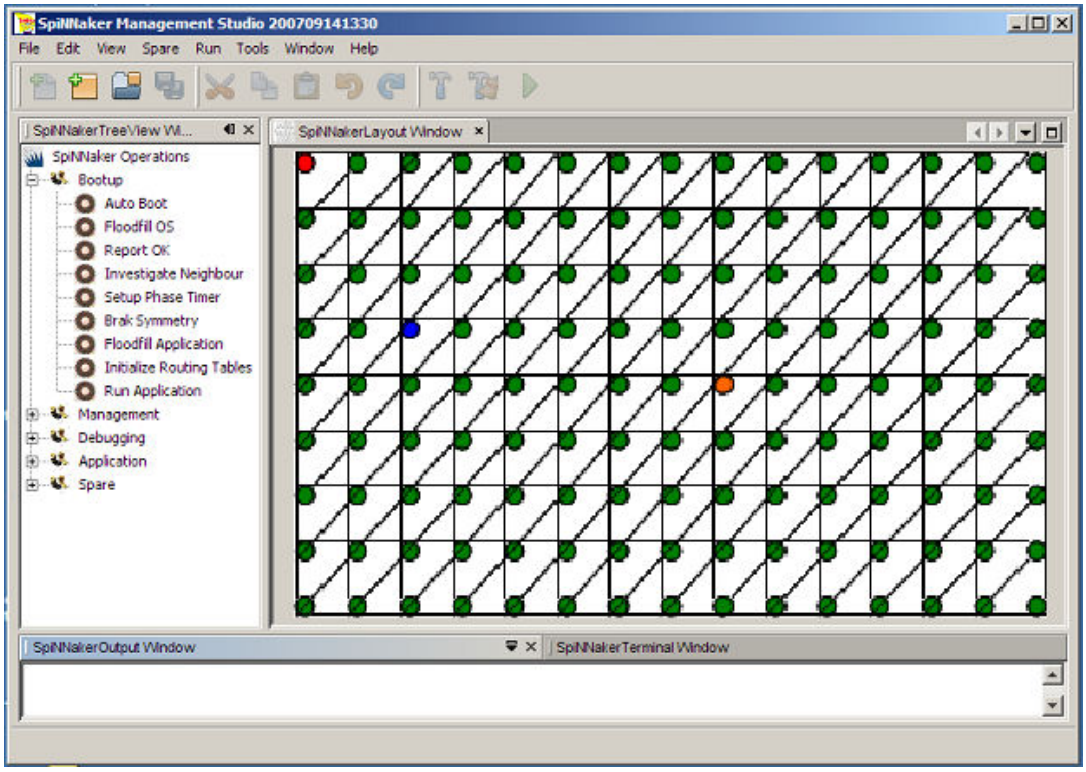

Figure 9.5: Host PC Graphical User Interface.

communication is explained in Appendix B. To handle the inter-CMP communication based on instructions from the Host PC, we use either NN or P2P packets to send a message to all the chips or specifically to a particular chip respectively. We use the address space 0x0F800000 to 0x0FFFFFFF in the routing key of the NN packet to convey a specific instruction. Some of the useful instructions used in the Ethernet frames and the P2P packets are listed in Appendix B.

9.4 Host PC User Interface

We are developing a user interface to be available on the Host PC. The purpose is to provide support for configuring the system i.e. boot-up, enquiring the state of the system, configuring the application based on the system-state before loading it into the system, loading the application into the system, and interactively managing the system/application for state reporting and fault-handling. It provides a graphical view of the SpiNNaker multi-CMP system for system management. The same interface can be used by users to communicate with the system. In the

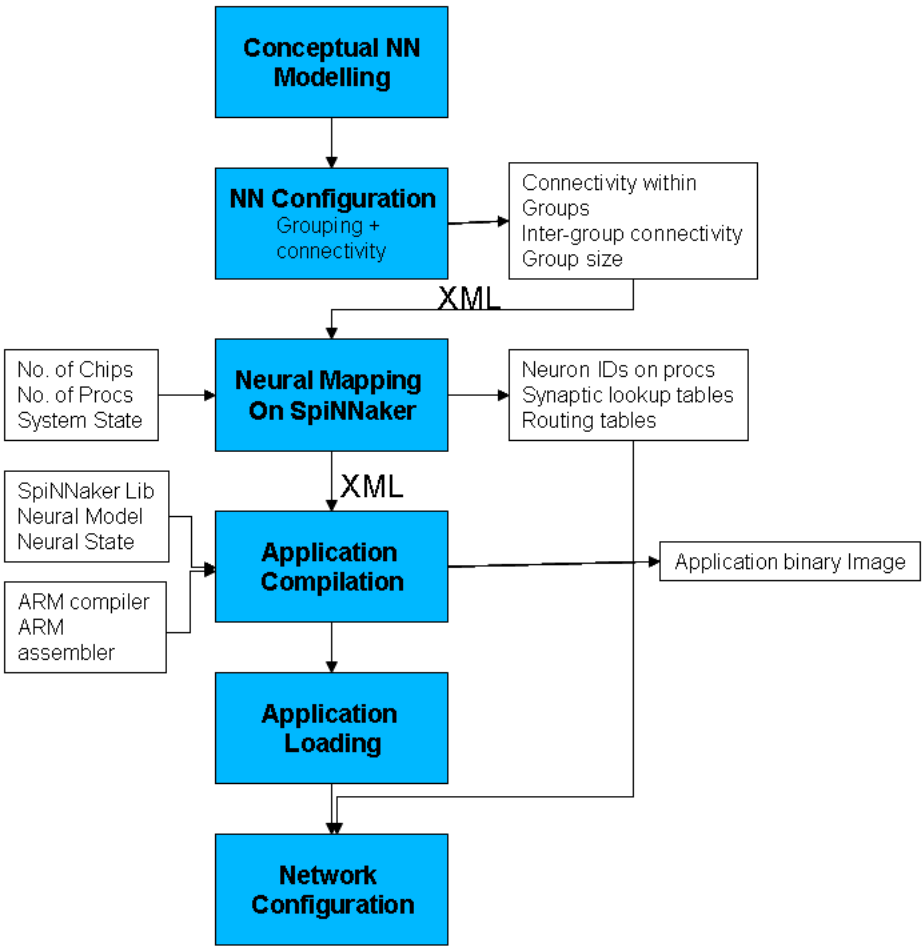

Figure 9.6: SpiNNaker Application Development Process.

context of a spiking neural simulation running in biological real time, the user may wish to provide run-time stimuli to the application and expect to receive responses. This can be done with the help of this interface. One of the purposes in providing this interface is to provide a local view of the system to the user or application developer whereby the user feels to be working on the local PC. The user's interface aims to show a graphical 2D view of the whole system as shown in Figure 9.5. The interface will connect itself with the SpiNNaker system using one or more Ethernet connections as explained in Chapter 4. The configuration application at the Host PC will run as part of the user's interface to interactively configure the system in phase II of the configuration process as explained in Section 7.2.

9.5 Application Development Process

The proposed process of application development for the SpiNNaker computing system is to develop a conceptual neural network application on a normal PC, configure it on the Host PC as per available resources (i.e. the number of CMPs in the system and number of processing nodes in each CMP) using a configuration application, and then to compile the neural simulation application based on these configuration files by "including" the SpiNNaker HAL library. The process is shown in Figure 9.6 with the input to various processes and their output. We use XML format for the configuration files describing the structure of the top-level neural network. The basic idea is to automate the process of the application development on a PC in a way which is seamless to the architectural differences. The configuration files take care of the following two aspects:

- Mapping the neurons onto the processing nodes distributed among the CMPs. The mapping should ensure load balancing among the processing nodes for application processing. Mapping should also group the neurons on the processing nodes so that neurons with co-related inputs ("fascicles") are grouped at the same processor to minimize traffic congestion.

- The routing tables in each CMP are configured taking into account the neural mapping onto the processing nodes. Two biologically co-located neurons may be placed at any location in the system, provided the routing tables have been properly configured to establish spike communication between them.

9.6 Summary

For easy application development on a hardware architecture and to provide support for an optimal use of its resources, it is important to provide a HAL for application developers to protect them from the underlying architectural details. HAL development does not follow a standard approach in the case of SoC design, especially in application-specific architectures using custom-designed components, which instead require the manual development of the HAL. The SpiNNaker SoC has been designed specifically to support optimal spiking neuron simulation and its expected users include researchers from multiple disciplines who would wish

to use it for developing and/or running neural simulations in real time. To facilitate varying interests, a HAL has been designed for the SpiNNaker computing system to facilitate application development and run-time interaction with the system. The SpiNNaker HAL includes functions to support chip-level boot up, system-level configuration, application loading, fault-handling routines, a Host-system communication mechanism, and other application support functions to enable developing neural applications for the SpiNNaker computing system. A systematic application development process is being proposed for SpiNNaker using a SpiNNaker API library in order to help developers of neural applications without being concerned too much about the underlying architecture, as well as to make optimal use of the designed features.

Chapter 10

Conclusions

This chapter summarises the dissertation and analyzes the proposed solutions in the light of the research objectives set forth at the start of this research. The strengths and limitations of the SpiNNaker configuration process, as proposed in this research, are presented along with certain aspects related to this research which could not be looked into due to the paucity of time. The chapter concludes with an envisaged implication of this research.

10.1 Dissertation Summary

In the initial chapters, a brief explanation of the neural dynamics contributing to the information processing inside neurons and inter-neuron communication was presented, along with the resulting combined neural network behaviour in the nervous system (Chapter 2). These two phenomena i.e. neural processing and the inter-neuron communication are the basis for designing a novel Application-Specific Integrated Circuit (ASIC) architecture in the form of the SpiNNaker massively-parallel neural net simulator. It is important to understand these phenomena to understand the SpiNNaker real-time application model devised to make optimal use of the SpiNNaker architecture. We also highlighted certain mysteries about the brain which require extensive exploration, such as the neural processing mechanism causing neurons to exhibit varying spiking behaviour in the presence of similar inputs, learning mechanisms as a result of temporal co-relation of inter-neuron communication, and the emergent properties of a population of neurons contributing to the stimulus/response behaviour from various

part of the nervous system. The thesis then focused on the need to create computer simulations for large populations of neurons (neural networks) (Chapter 3) to understand neural information processing and the spatio-temporal behaviour of these networks. A few software- and hardware-based approaches to simulate large-scale neural populations from the literature were analysed highlighting their limitations which motivated the building of a specific-to-purpose large-scale hardware simulation engine for better performance. The thesis discussed the need for designing a scalable and high-performance neural network simulator with specific features to support distributed parallel processing inside each neuron and the massive inter-neuron communication. The objectives in designing the SpiNNaker computing system were highlighted along with a description of its architecture. The SpiNNaker application model to support large-scale neural simulations on the SpiNNaker computing system was also explained (Chapter 4). Some users' expectations were captured which provided a motivation for this research to facilitate the application development and the system's use to make it useable by a variety of potential users.

In the later part of this dissertation, the focus was on the research work performed to devise a mechanism to support real-time neural applications on the SpiNNaker massively-parallel computing system. The need for creating a system-level model of the SpiNNaker computing system was explained in Chapter 5, to verify the proposed configuration mechanism and to develop and test neural applications for SpiNNaker. Our experience while creating a novel complete-system model for a multi-CMP system to an instruction- and cycle-accurate level were described, along with a description of the process adopted to verify the system-level model with the help of case studies. This model is one of the most important contributions of this research, as not only did it provide a verification platform to validate the SpiNNaker configuration process but it also was very useful in validating the design objectives of the SpiNNaker computing system. The SpiNNaker system-level model is being used for application development for SpiNNaker and verification of the hardware components. The configuration process for two multi-CMP systems was presented in Chapter 6 to capture the configuration issues and some implemented solutions to large-scale multi-CMP computing systems. We also presented a few important peculiarities of the SpiNNaker computing system which invalidated the configuration process that is used

in most multi-CMP systems. The dissertation, then, presented the proposed configuration and application loading process in Chapter 7 after highlighting some configuration challenges specific to the SpiNNaker architecture in line with the ones already described in Chapter 6. Experimental results were presented to verify that the research objectives had been met. One chapter (Chapter 8) was dedicated to the introduction of the fault-tolerance features of the SpiNNaker configuration process to make SpiNNaker a reliable computing system. At the end of this dissertation, some details of the SpiNNaker Hardware Abstraction Layer (HAL), developed to help programmers in application development for the SpiNNaker computing system were presented (Chapter 9). A few features of a proposed user interface to interact with the system were also presented as ongoing work.

10.2 Research Analysis

10.2.1 Response to Research Objectives

In response to the research objectives outlined in Chapter 1 Section 1.3, the following describes how this research has tried to meet its objectives:

- In response to objective of efficiently testing and configuring all the devices in each CMP of the SpiNNaker computing system to enable running an application on it, Chapter 6 shows that it takes only about ∼1.3 ms to test all the chip devices and configure them to the default setting optimal for the standard SpiNNaker application model. The time is independent of the number of CMPs in the SpiNNaker system as the process of initialization is run concurrently by all the CMPs from their Boot ROMs. The process is almost independent of the number of processing nodes inside a CMP as the code is loaded into each processor's tightly coupled memory before execution and run concurrently until the monitor processor is selected. A slight delay is caused by more processing nodes for contention to access Boot ROM while loading the code to the local memory, however, this delay is not substantial. After its selection, only the monitor processor in each CMP configures the remaining chip resources while the other (application) processors are in sleep mode. This avoids contention on the shared communication medium (System-NoC) and the shared memory (Boot ROM),

thus making the process quite efficient.

- In response to the question of configuring a multi-CMP system externally at run-time to make it to work as one integrated system using the homogeneous CMP-interconnect fabric to be used by the application, a novel multi-CMP configuration process was proposed in Chapter 6. The process configures all the CMPs in the system to collaborate in an integrated system establishing inter-CMP communication and the SpiNNaker system's communication with the Host PC. The process uses the SpiNNaker event-driven application model with the help of nearest neighbour packets over the asynchronous system interconnect for this purpose. A novel communication protocol using nearest neighbour packets has been devised to manage the SpiNNaker configuration process with the help of functions embedded into each CMP's Boot ROM.

- We tackled the objective of loading a tailored neural simulation application into the SpiNNaker system, in a scalable way, in Chapter 6. A novel asynchronous application loading process has been presented, which uses the SpiNNaker event-driven application model. We have shown empirically that the process is quite efficient and scalable (i.e. the process is virtually independent of the size of the system in a large-scale multi-CMP system). As per the proposed process, a pipelined wave of application packets fan-out from the Host-connected chip(s) to wrap the whole system, whereby loading a typical application of 100-KB in less than $\sim$20 ms, irrespective of the size of the system.

- The question of supporting neural simulation applications using the SpiNNaker real-time event-driven application model over the SpiNNaker multi-CMP system was answered in Chapter 9. The SpiNNaker Hardware Abstraction Layer (HAL) was devised as part of this research to support application development for the SpiNNaker computing system. During the CMP initialization process, we configure each processing node to run the SpiNNaker event-driven application model with the help of template Interrupt Service Routines (ISRs) with a user-application entry point in each ISR.

- The objective of enabling users to communicate interactively with the system and the application for diagnostics or real-time interactive application support was looked into in Chapter 9. The process uses the Ethernet connection between the Host PC and the SpiNNaker system for the communication between the user interface on the Host PC and the application running on the SpiNNaker multi-CMP system. The monitor processor on the Host-connected CMP transforms the Ethernet frame-based communication into small P2P packets which are then forwarded to the destination chip's monitor processor. This way, monitor processors on all the chips contribute to exhibit an integrated system-wide view of the whole SpiNNaker multi-CMP system on the Host PC. The proposed user interface is intended to diagnose each individual CMP and its devices interactively with the help of a Graphical User's Interface (GUI). The same procedure is used to communicate with the application at run-time to view the application state or to communicate with the application to provide the stimuli to it and get the responses back.

- On how to make the configuration process and application execution process fault-tolerant both at CMP- and system-level, we presented the fault-tolerance features implemented as part of the SpiNNaker configuration process in Chapter 8. We evaluated the proposed features empirically for their real-time performance and robustness. The configuration process was devised with the aim to minimize single points-of-failure at each stage of its execution, however, if such a situation arises, the fault-recovery functions are use to recover the system in minimum possible time without interfering with the neural application.

10.2.2 Strengths of the Configuration Process

- No Separate Configuration Network: In contrast to the other multi-CMP systems, such as Blue Gene(L) or Cray XT-3 (Chapter 6), the SpiNNaker multi-CMP system does not use a separate dedicated network to configure the system and to load the application into each CMP. We use the same homogeneous inter-CMP fabric for this purpose, which is later used by the application for inter-process communication. This reduces the overhead of

maintaining another network alongside SpiNNaker's internal network, alleviating the need for extra switches, wires and other network infrastructure to be used for this purpose. The fact that the SpiNNaker packet switching Communication Network is very efficient as compared to an Ethernet connecting all CMPs to the Host PC makes the application loading process much faster. The process makes use of the broadcast feature of the nearest neighbour packet along with the router's capability to broadcast this type of packet to all six neighbours in only one router cycle. Moreover, it does not require any network configuration to start the flood-fill process as the router has been hard-wired to handle NN packets.

- Performance: The SpiNNaker configuration process has been optimized for performance and code size. The code has been written in ARM assembly language with optimal instructions at the device level. The process uses the same asynchronous event-driven application model right from chip initialization, that is used by the SpiNNaker application itself. By doing this the application template is already loaded and running in the system and the remaining application, once loaded, is easily plugged into the template. This avoids the need for separate code for initialization which otherwise would be useless once the application is loaded. The device drivers have been written for optimal use of all of the components' features.

- Scalability: The configuration and application loading process makes full use of the parallel and hierarchical architecture of the SpiNNaker computing system, thereby making the process very scalable. Each CMP contains a local copy of the code which is copied by each processor to its local memory, making the configuration process independent of the number of processors in each CMP and the number of CMPs in the SpiNNaker multi-CMP system. A perfectly-pipelined application-loading mechanism propagates waves of data emanating from the Host-connected CMP to all the chips which makes the process virtually independent of the size of the system. Network congestion may affect the performance; however, adopting a selective flood-fill process in the forward direction (Chapter6) can help overcome this problem.

- Fault-tolerance: The proposed mechanism makes full use of the fault-tolerance features designed specifically in the SpiNNaker computing system to make

the system highly fault-tolerant at run-time. The process itself has been devised to be robust enough to succeed in the wake of envisaged hardware faults. We attempt to minimize single points-of-failures in the process and provide fault-handling routines to cope with likely hardware faults. The process has been designed to bring faulty components back into service or isolate them to avoid disruption to the other components. The objective is to bring each SpiNNaker CMP up despite a small number of faulty processing nodes, and to bring the SpiNNaker system up despite a few dead (or partially dead) CMPs.

- Real-time Host-system Interaction Support: The proposed mechanism implements an interactive Host-system communication protocol whereby the Host PC can communicate with the whole system at run-time to externally configure the system and load an application into it. The protocol has been refined to enable the Host PC to communicate with any individual CMP in the system to diagnose its state. This mechanism can be used to provide a user-system or user-application run-time interaction for system- or application-level management.

- Application Development Support: A by-product of the proposed configuration mechanism is a library of useful functions compiled to help software development for the SpiNNaker computing system. The library has been organized with a hierarchy of functions to support the development of neural applications without worrying about the underlying devices in the system or its architecture. We have organised the library into a hierarchy of functions from low-level device handling functions to application-level constructs to help developers at multiple levels.

- User Interface: Work is still ongoing on a high-level graphical user interface (GUI) for the Host PC to interact graphically with the system at run-time. The GUI intends to provide many automated features to support run-time application configuration, diagnostic/debugging, real-time system reconfiguration, and application state reporting. The purpose is to provide a user with a feeling of working on a PC by virtualizing the low-level functional details.

10.2.3 Limitations

- Global Synchronization: The system is highly asynchronous with independent clocks and asynchronous inter-chip communication. This is quite expected of a large distributed system. However, we need to have some kind of system-wide synchronisation for certain application requirements as explained in Chapter 5. This is very difficult to achieve with no global clock in the system and without a central synchronisation mechanism. The proposed mechanism is to synchronise the system at a coarse granularity of a few milliseconds with the help of NN packets broadcast to all the CMPs (just as in the flood-fill process) after a certain period during application execution. The mechanism has not yet been fully implemented in the absence of a full-scale application.

- Run-time Application State: Many neural network applications, especially ones based on multi-layer perceptron (MLP) neural networks, require the state of the application to be reported to the user after a certain period of its execution, as already explained as a user's expectation in Chapter 5. The state of a certain number of neurons may also need investigating for debugging purposes. This is, however, very difficult to acquire in a large-scale dynamic distributed system. We can timestamp the state of the system at certain points in time, however, in the absence of a global clock there may be a timing skew. Moreover, getting a huge amount of data from the system to the Host PC for viewing the state of the neural network may be a very time-consuming process, interfering with the run-time application during its inter-process communication.

- Limited Fault-handling: We have tried to handle most expected hardware faults, however, this does not mean that the process can handle all faults occurring at run-time. Very simple fault-handling routines have been written in view of the limited memory in the monitor processor. Our aim is to start with very simple solutions and then to refine the fault-tolerance mechanism in the course of system evolution. The fact that SpiNNaker has been designed as a fully reconfigurable system, which can be externally diagnosed and reconfigured at run-time, allows us to keep a very simple first layer of fault-tolerance within the system while employing a more sophisticated mechanism interactively at run-time.

- Limited Management Support: We intend to provide the system state in a graphical form to the user at the Host PC. However, this feature is still not fully implemented and will be refined with the passage of time. This is being looked into as a separate research project as part of the SpiNNaker research project.

- Lack of Dynamic Reconfiguration: Neural networks reconfigure their connectivity based on run-time learning. The SpiNNaker system provides support whereby the network can be reconfigured internally or externally to change the neural connectivity at run-time. The configuration process does not, however, provide high-level support for this feature at the moment. The problem is being looked into as a separate PhD research project in the SpiNNaker research group.

10.3 Suggested Future Directions

As described before, the configuration process proposed in this research is not a perfect solution and does allow room for improvement. A few research aspects could not be looked into due to the paucity of time, as was mentioned in Section 10.2.3 of this chapter. The work as part of the SpiNNaker research project is still in progress on areas such as the user interface at the Host PC, adaptive neural network reconfiguration at run-time based on neural learning dynamics, run-time user-application interaction to enable reporting of the state of application to the user, global synchronization for routing and application support, and system-level diagnostics including network performance monitoring. SpiNNaker network management and dynamic network reconfiguration are ongoing PhD research topics as part of the SpiNNaker research group, while the work on the SpiNNaker user interface is being carried out in collaboration with the SpiNNaker application group at the University of Southampton. Similarly, the neural application mapping along with the underlying neural network configuration is being looked into by the University of Southampton. Besides this, we are trying to provide a few sample spiking neural network implementations as part of the user interface as a tutorial to motivate users to work on the SpiNNaker neural network simulator. The work is being done as a separate PhD project.

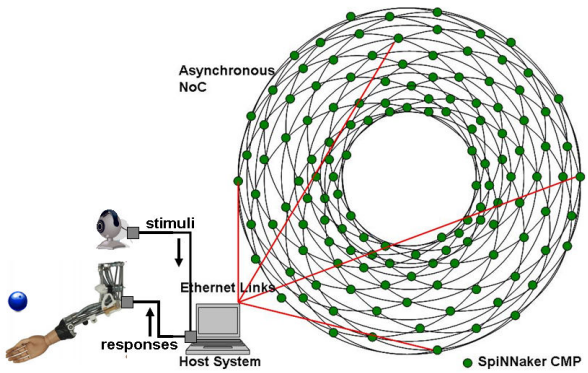

Figure 10.1: Example Real-time Interactive Neural Application on SpiNNaker Controlling a Robotic Arm.

10.4 Research Implications

This research is a starting point towards building an interactive real-time neural application for the SpiNNaker computing system. The purpose of such interactive applications is to explore the mammalian nervous system at various levels of detail, starting from understanding the information processing mechanism in individual neurons to understand the emergent behaviour of neural networks. The SpiNNaker configuration and application support mechanism, proposed in this dissertation, can support a variety of neural applications during their development, configuration and execution. Figure 10.1 shows an envisaged interactive application model in which a neural application running on SpiNNaker is responding to stimuli received from a web-cam and passing back responses to a robotic arm to control a ball. The user-interface on the Host PC interacts in two directions, i.e. receiving the input from the web-cam to pass to the SpiNNaker system and receiving the responses from the system to pass to the robotic hand. The neural application does real-time processing of these inputs to formulate a response to be sent back as output. The response generated at a certain location of a multi-CMP system would be transported to the Host-connected chip in the form of multicast packets, which would pass it to the Host PC as Ethernet frame(s). A similar process can take place to control a hominoid robot with the help of a "remote brain" being simulated by the SpiNNaker computing system. The application-support-process proposed in this dissertation is expected to provide useful execution support to such applications with the help of the SpiNNaker HAL functions.

Bibliography

[ABBea03] G. Almasi, R. Bellofatto, J.e. Brunheroto, and et al. An Overview of the BlueGene(L) System Software Organization. In *Euro-Par '03 Conference on Parallel and Distributed Computing*, 2003.

[ABCea05] N. R. Adiga, M. A. Blumrich, D. Chen, and et al. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49:289–301, 2005.

[AKB+07a] S. R. Alam, J. A. Kuehn, R. F. Barrett, J. M. Larkin, M. R. Fahey, R. Sankaran, and P. H. Worley. Cray XT4: An early evaluation for petascale scientific simulation. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC '07)*, November 2007.

[AKB+07b] S. R. Alam, J. A. Kuehn, R. F. Barrett, J. M. Larkin, M. R. Fahey, R. Sankaran, and P. H. Worley. UltraSPARC T2: A Highly-Threaded, Power-Efficient, SPARC SoC. In *Proceedings of Asian Solid-State Circuirts Conference (A-SSCC 2007)*, November 2007.

[All00] Visual Socket Interface Allience. *System-Level Interface Behavioral Documentation Standard (SLD 1 1.0)*. VSIA, sld 1 1.0:2000 edition, Mar. 2000.

[AM00] Lars Albertsson and Peter S Magnusson. Using Complete System Simulation for Temporal Debugging of General Purpose Operating Systems and Workloads. In *In Proceedings of Mascots 2000*, pages 191–198. Society Press, 2000.

[AS96] J. Torrellas A. Sharma, A.T. Nguyen. *Augmint - a Multiprocessor Simulation Environment for Intel x86 Architectures - Technical*

Report. Center for Supercomputing Research and Development - University of Illinois at Urbana-Champaign, 1996.

[BA97] Doug Buger and Todd M. Austin. The SimpleScaler Tool Sset, Version 2.0. *The University of Wisconsin-Madison Computer Sciences Department Technical Report no. 1342*, June 1997.

[BKM+00] L.A. Barroso, K.Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 282–293, 2000.

[BMW08] Simeon A. Bamford1, Alan F. Murray, and David J. Willshaw. Large Developing Axonal Arbors Using a Distributed and Locally-Reprogrammable Address-Event Receiver. In *Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008)*, 2008.

[CACY+06] Isci Canturk, Buyuktosunoglu Alper, Cher Chen-Yong, Bose Pradip, and Martonosi Margaret. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, Washington, DC, USA, 2006. IEEE Computer Society.

[CFBC06] P. Crowley, M. A. Franklin, J. Buhler, and R. D. Chamberlain. Impact of CMP Design on High-Performance Embedded Computing. In *Proc. of 10th High Performance Embedded Computing Workshop*, pages 33–34, September 2006.

[Com06] IEEE Design Automation Standards Committee. *IEEE Standard SystemC Language Reference Manual*. IEEE Computer Society, std. 1666-2005 edition, Mar. 2006.

[DA01] Peter Dayan and L.F. Abbott. *Theoretical Neuroscience*. MIT Press, Cambridge, 2001.

[ea03] L. Ceze et al. Full Circle: Simulating Linux Clusters on Linux Clusters. In *Fourth LCI International Conference on Linux Clusters The HPC Revolution*, 2003.

[ea08] S. R. Alam et al. An Evaluation of the Oak Ridge National Laboratory Cray XT3. *International Journal of High Performance Computing Applications archive*, 22:52–80, February 2008.

[EKR06] R. Eickhoff, T. Kaulmann, and U. Rückert. SIRENS: A Simple Reconfigurable Neural hardware Structure for Artificial Neural Network Implementations. In *Proc. 2006 Int'l Joint Conf. Neural Networks (IJCNN2006)*, pages 2830–2837, 2006.

[EMIE04] J. A. Gally E. M. Izhikevich and G.M. Edelman. Spike-Timing Dynamics of Neuronal Groups. *Cerebral Cortex*, 14(8):933–944, 2004.

[FSM08] Johannes Fieres, Johannes Schemmel, and Karlheinz Meier. Realizing Biological Spiking Network Models in a Configurable Wafer-Scale Hardware System. In *Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008)*, 2008.

[FT07] Steve Furber and Steve Temple. Neural Systems Engineering. *J. R. Soc. Interface*, 4(13):193–206, April 2007.

[Fur05] S. B. Furber. Future Trends in SoC Interconnect. In *IEEE International Symposium on Design, Automation and Test (VLSI-TSA-DAT)*, pages 290–293, April 2005.

[GBea05] A. Gara, M. A. Blumrich, and D. Chen et al. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49:289–301, 2005.

[Ghe05] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded System*. Springer Publishers, New York, NY, USA, 2005.

[GID06] E. Chicca G. Indiveri and R. Douglas. A VLSI Array of Low-power Spiking Neurons and Bistable Synapses with Spike-Timing Dependent Plasticity. *IEEE Trans. Neural Networks*, 17(1), January 2006.

[Gre08] G.W. Greenwood. Attaining Fault Tolerance through Self-adaption: The Strengths and Weaknesses of Evolvable Hardware

Approaches. In *Proc. 2008 World Congress on Computer Intelligence (WCCI2008)*, pages 368 – 387. Springer-Verlag Berlin Heidelberg, 2008.

[Gro02] T. Grotker. *Computer Simulation Using Particles*. Kluwer Academic Publishers, Boston, 2002.

[HAM07] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward Neural Network Implementation in FPGA Using Layer Multiplexing for Effective Resource Utilization. *IEEE Trans. Neural Networks*, 18(3):880–888, May 2007.

[HBB+05] R. A. Haring, R. Bellofatto, A. A. Bright, P. G. Crumley, M. B. Dombrowa, S. M. Douskey, M. R. Ellavsky, B. Gopalsamy, D. Hoenicke, T. A. Liebsch, J. A. Marcella, and M. Ohmacht. Blue Gene/L compute chip: Control, test and bring up intrastructure. *IBM Journal of Research and Development*, 49:289–301, 2005.

[HE88] Roger W. Hockney and James W. Eastwood. *Computer Simulation Using Particles*. CRC Press, New York, 1988.

[Her96] S. Herrod. *The simOS Simulation Environment - Technical Report*. Computer Systems Laboratory - Stanford University, 1996.

[HG93] B.A. Huberman and N.S. Glance. Evolutionany Games and Computer Simulations. *National Academy of Science USA*, 90:7716–7718, August 1993.

[HH52] A. L. Hodgkin and A. F. Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *Journal of Physiology*, 117(4):500544, 1952.

[HLWW02] Anssi Haverinen, Maxime Leclercq, Morman Weyrich, and Drew Wingard. *White Paper - SystemC Based SoC Communication Modeling for the OCP Protocol*. www.ocpip.org, v1.0 edition, Oct. 2002.

[HNO97] L. Hammond, B. A. Nayfeh, and K. Olukotun. A Single Chip Multiprocessor. *IEEE Computer Magazine*, 30:79–85, April 1997.

[HSB04] Jeff Hawkins and Sandra Blakeslee. *On Intelligence*. Times Books, Henry Holt and Company, New York, 2004.

[IE08] E. M. Izhikevich and Gerald M. Edelman. Large-scale Model of Mammalian Thalamocortical Systems. *Proceedings of the National Academy of Sciences*, 105(9):cmp3593–3598, March 2008.

[Inc05a] Cray Inc. *Cray X1E Datasheet*. http://ed-thelen.org/ comp-hist/ CRAY-1- HardRefMan/ CRAY-1-HRM.html, January 2005.

[Inc05b] Cray Inc. *Cray XT3 Datasheet*. Cray Inc., http://www. craysupercomputers. com/ downloads/ CrayXT3/ CrayXT3_ Datasheet. pdf, January 2005.

[Izh03a] E.M. Izhikevich. Simple Model of Spiking Neurons. *IEEE Trans. on Neural Networks*, 14:1569–1572, November 2003.

[Izh03b] E.M. Izhikevich. Which Model to Use for Neocortical Spiking Neuron. *IEEE Trans. on Neural Networks*, 14:1569–1572, November 2003.

[Izh07] E. M. Izhikevich. *Dynamical Systems in Neuroscience: The Geometry of Excitability and Bursting*. The MIT Press, Cambridge Massachusetts, London England, 2007.

[JFW08] X. Jin, S.B. Furber, and J.V. Woods. Efficient Modelling of Spiking Neural Networks on a Scalable Chip Multiprocessor. In *Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008)*, 2008.

[Joh06] Long John. *Comprehensive SystemC Training - Training Manual*. Doulos UK., http://www.doulos.com/, 2006.

[JSR+97] A. Jahnke, T. Schönauer, U. Roth, K. Mohraz, and H. Klar. Simulation of Spiking Neural Networks on Different Hardware Platforms. In *7th Int'l Conf. on Artificial Neural Networks (ICANN '97)*, 1997.

[KJFP07] M. Khan, X. Jin, S. Furber, and L.A. Plana. System-Level Model for a GALS Massively Parallel Multiprocessor. In *Proc. 19th UK Asynchronous Forum*, pages 9–122, September 2007.

[KLP+08] M.M. Khan, D.R. Lester, L.A. Plana, A. Rast, X. Jin, E. Painkras, and S.B. Furber. Spinnaker: Mapping Neural Networks onto a Massively-Parallel Chip Multiprocessor. In *Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008)*, 2008.

[KNJ+08a] M.M. Khan, J. Navaridas, X. Jin, L.A. Plana, J.V Woods, and S.B. Furber. Configuring a GALS CMP System for Real-time Applications. In *Proc. 20th UK Asynchronous Forum*, pages 9–122, September 2008.

[KNJ+08b] M.M. Khan, J. Navaridas, X. Jin, L.A. Plana, J.V Woods, and S.B. Furber. Real-Time Application Support for a Novel SoC Architecture. In *Proc.* 4^{th} *UK Embedded Forum*, Southampton, UK, September 2008.

[KNR+09] M.M. Khan, J. Navaridas, A.D. Rast, X. Jin, L.A. Plana, M. Luján, J.V. Woods, J. Miguel-Alonso, and S.B. Furber. Event-Driven Configuration of a Neural Network CMP System over a Homogeneous Interconnect Fabric. In *In Proceedings of International Symposium on Parallel and Distributed Computing (ISPDC2009)*, June 2009.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[LBHS06] Yingmin Li, David Brooks, Zhigang Hu, and Kevin Skadron. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 71–82, 2006.

[LET97a] ED Lumer, GM Edelman, and Tononi. Neural dynamics in a model of the thalamocortical system. I. Layers, loops and the emergence of fast synchronous rhythms. *Cereb. Cortex*, 7(3):207–227, 1997.

[LET97b] ED Lumer, GM Edelman, and Tononi. Neural dynamics in a model of the thalamocortical system. II. The role of neural synchrony tested through pcmperturbations of spike timing. *Cereb. Cortex*, 7(3):228–236, 1997.

[LLB+06] Y. Li, B. Lee, D. Brooks, Z. Hu, and K. Skadron. CMP Design Space Exploration Subject to Physical Constraints. In *In 12th International Symposium on High Performance Computer Architecture (HPCA-12)*, 2006.

[LM06] J. Li and J. Martinez. Dynamic Power-Performance Adaptation of Parallel Computation on Chip Multiprocessors. In *In Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA-12)*, 2006.

[Ltd08a] ARM Ltd. Arm968E-S Processor. ARM Official Website, May 2008. http:// www. arm. com/ products/ CPUs/ ARM968E-S. html, accessed on 31 May 2008.

[Ltd08b] Virtutech Ltd. *SIMICS: Virtualized Software Development*. http:// www. virtutech. com/, San Jose, CA 95110 USA, 4.0 edition, October 2008.

[Mar06] Henry Markram. The Blue Brain Project. *Nature Reviews Neuroscience*, 7:153–160, February 2006.

[MBH+05] Michael R. Marty, Jesse D. Bingham, Mark D. Hill, Alan J. Hu, Milo M.K. Martin, and David A. Wood. Improving Multiple-CMP Systems Using Token Coherence. In *In Proceedings of the 11th International Symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[Me98] W. Maass and C. M. Bishop (ed). *Pulsed Neural Networks*. MIT Press, Cambridge, Massachusetts, 1998.

[MLFS97] H. Markram, J. Lubke, M. Frotscher, and B. Sakmann. Regulation of synaptic efficacy by coincidence of postsynaptic aps and epsps. *Science*, 213-215:153–160, 1997.

[MM00] Robert M. McGraw and Richard A. MacDonald. Abstract Modeling for Engineering and Engagement Level Simulations. In *WSC '00: Proceedings of the 32nd conference on Winter simulation*, pages 326–334, San Diego, CA, USA, 2000. Society for Computer Simulation International.

[New06] BBC News. Scientists to Build 'Brain Box', July 2006. `http://news.bbc.co.uk/1/hi/england/manchester/5187596.stm`.

[OBea05] M. Ohmacht, R. A. Bergamaschi, and S. Bhattacharya et al. Blue Gene/L compute chip: Memory and Ethernet subsystem. *IBM Journal of Research and Development*, 49:289–301, 2005.

[Org09] TCP DUMP Org. Tcp dump libpcap. Website Resource, May 2009. http:// www. tcpdump. org/, accessed on 19 May 2009.

[Pan01] P.R. Panda. SystemC A Modeling Platform Supporting Multiple Design Abstractions. In *ACM ISSS 01*, Montreal Quebec, Canada, 2001.

[PBF+08] L.A. Plana, J. Bainbridge, S. Furber, S. Salisbury, Y. Shi, and J. Wu. An On-Chip and Inter-Chip Communications Network for the Spinnaker Massively-Parallel Neural Net Simulator. In *Proc. Second ACM/IEEE International Symposium on Networks-on-Chip (NoCS 2008)*, pages 215 – 216, 2008.

[PBK91] D. Probert, J. L. Bruno, and M. Karaorman. SPACE: A New Approach to Operating System Abstraction. In *In International Workshop on Object Orientation in Operating Systems*, pages 133–137, 1991.

[PFT+07] Luis A. Plana, Steve B. Furber, Steve Temple, Mukaram Khan, Yebin Shi, Jian Wu, and Shufan Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design & Test of Computers*, 24(5):454–463, Sept.-Oct. 2007.

[Pro07] The SpiNNaker Project. *SpiNNaker - a Chip Multiprocessor for Neural Network Simulation.* The University of Manchester, 0.5 (draft) edition, Nov. 2007.

[PWKR02] M. Porrmann, U. Witkowski, H. Kalte, and U. Rückert. Implementation of Artificial Neural Networks on a Reconfigurable Hardware Accelerator. In *Proc. 2002 Euromicro Conf. Parallel, Distributed, and Network-based processing*, pages 243–250, 2002.

[RCAT97] G. Lindzey R. C. Atkinson and R. F. Thomspon, editors. *How Brain Thinks - Evolvig Intellegence, Then and Now.* Wiedenfeld & Nicolson, The Orien Publising Group, London, 1997.

[RKJ+09] A.D. Rast, M.M. Khan, X. Jin, L.A. Plana, and S.B. Furber. A Universal Abstract-Time Platform for Real-Time Neural Networks. In *Proc. 2009 Int'l Joint Conf. on Neural Networks (IJCNN2009)*, 2009.

[RLT78] B. Randell, P. Lee, and P. C. Treleaven. Reliability Issues in Computing System Design. *ACM Comput. Surv.*, 10(2):123–165, 1978.

[Roh99] D. L. T. Rohde. LENS: The light, efficient network simulator version 2.63. *http://tedlab.mit.edu/ dr/Lens/ - Retrieved 17 June 2007*, 1999.

[Ros95] M. Rosenblum. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 1995.

[RSV87] R.L. Rudell and A. Sangiovanni-Vincentelli. Multiple-Valued Minimization for PLA Optimization. *IEEE Trans. on Computer-Aided Design*, 6(5):727–750, September 1987.

[RYKF08] A.D. Rast, S. Yang, M. Khan, and S.B. Furber. Virtual Synaptic Interconnect Using an Asynchronous Network-on-Chip. In *Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008)*, 2008.

[Swi08] EPFL Switzerland. Blue Brain Project. *www source*, 2008.

[SY03] Ahmed A. Jerraya Sungjoo Yoo. Introduction to Hardware Abstraction Layers for SoC. In *in Proc. of Design, Automation and Test in Europe Conference and Exhibition (DATE'03)*, page 10336, march 2003.

[TDF+02] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr. H. Le, and B. Sinharoy. Power4 System Microarchitecture. *IBM Journal for Research and Development*, 46:1–25, January 2002.

[Tho00] R. F. Thomspon(ed). *The Brain - A Neuroscience Primer.* W.H.Freeman and Company, Worth Publisher, New York, 3 edition, 2000.

[TMM+06] F. Tuffy, L. McDaid, M. McGinnity, J. Santos, P. Kelly, V. W. Kwan, and J. Alderman. A Time-Multiplexing Architecture for Inter-Neuron Communications. In *Proc. 2006 Int'l Conf. Artificial Neural Networks (ICANN 2006)*, pages 944–952, 2006.

[Tra02] Thomas P. Trappenberg. *Fundamentals of Computational Neuroscience*. Oxford University Press, New York, 2002.

[Wik09a] Wikipedia. Axon. Website Resource, Feb 2009. http://en.wikipedia.org/wiki/Axon, accessed on 13 Feb 2009.

[Wik09b] Wikipedia. Computer. Website Resource, Feb 2009. http://en.wikipedia.org/wiki/Computer, accessed on 13 Feb 2009.

[Wik09c] Wikipedia. Hp computer. Website Resource, Feb 2009. http://docs.hp.com/en/32650-90871/go01.htm, accessed on 13 Feb 2009.

[Wik09d] Wikipedia. Resting potential. Web Resource, Feb 2009. http:// faculty. stcc.edu/AandP/AP/AP1pages/nervssys/unit10/resting. htm, accessed on 13 Feb 2009.

Appendix A

SpiNNaker Inter-CMP NN Communication Protocol

A.1 Introduction

This appendix describes the instructions used with nearest neighbour (NN) packets to facilitate phase II of the boot-up process (system-level integration) and the application loading (flood-fill) process. The SpiNNaker address space 0x0F800000-0x0FFFFFFF has been dedicated to be used for specific instructions in the NN packet's routing key. An NN packet (Chapter 4 Figure 4.5) with NN-route type set as "normal" and containing 0x0F8 in bits 31-20 of its routing key is treated by the monitor processor at the recipient chip as an instruction packet or a response to its previous instruction. An NN packet with routing key contents other than this range is taken to carry data in its payload and a SpiNNaker CMP physical address in its routing key as part of the application loading flood-fill process as explained in Chapter 7. The instructions cover those to request chip status, report chip status, assign chip addresses, start the flood-fill process, and to control the communication between the Host PC and a specific chip in the system during system-level configuration. As part of application loading process, this protocol includes instructions to serialize the data, control the flow of data, request missing data, and some other configuration instructions. We have also devised instructions to control the nearest neighbour diagnostic and recovery process as explained in Chapter 8. The following sections list some of these instructions:

A.2 System-level Configuration Process

Table A.1 gives the instructions used in the system-level configuration process with their meaning given in front of each instruction:

Table A.1: System-level Configuration Instructions.

Value	**Instruction**	**Meaning**
221	NN_CONF_SYS_ SIZE	Size of system is in the packet's payload
222	NN_CONF_USE_TIME_ PHASE	Timephase duration (number of milliseconds) to be used for counting timephase for the router. The value is in the packet's payload.
223	NN_CONF_RESET_TIME_ PHASE	reset the time phase to 0.
224	NN_CONF_BREAK_ SYMMETRY	Break the symmetry, i.e. the chip should compute its relative address and assign it to itself, the sender's address in the payload and the direction of sender in the "NN-route" field of the control-field.
225	NN_CONF_SEND_OK_ REPORT	Request to send short status report to the Host-Connected chip
226	NN_CONF_REPORT_ ALL_FINE	It is a status report from a chip indicating the chip is OK.
227	NN_CONF_REPORT_ WITH_STATUS	Request to send a detailed status report to the Host-Connected chip
228	NN_CONF_BOOT_INTRS_ DONE	Extra boot instructions loading completed, start running from the address in the payload
229	NN_CONF_APP_LOAD_ DONE	Application download completed, start executing it from the address in the payload

A.3 NN Diagnostic Process

Table A.2 lists the instructions used as part of nearest neighbour diagnostic and recovery process as explained in Chapter 8:

Table A.2: NN Diagnostic and Recovery Instructions.

Value	**Instruction**	**Meaning**
231	NN_DIAG_HELLO	A message from a neighbour to others, "hello i am alive". No payload with this packet. The sender's location is received by the receiver in the "NN-route" field of the packet's control. No response is expected by the sender.
232	NN_DIAG_HOW_R_U	A message from a neighbour to others, "how are you"? The sender expects a response back i.e. a response with "hello i am alive" from that particular neighbour.
233	NN_DIAG_HAVE_U_ HEARD_OUR_N	A message from a neighbour to others, "have you heard from my neighbour on your link X". (link X is given in the Routing key bits 11-0). No payload is attached. The sender expects a response.
234	NNDIAG_DEAD_CHIP_ WHO_TO_CURE	A message from a neighbour to others, "Chip number X to do the nurse chip". Number X is given in the Routing key bits 11-0.
235	NN_DIAG_YES_ I_HEARD	A message from a neighbour to an other chip, "yes i have heard from our common neighbour". Sent as a response to a packet asking the health of a common neighbour.
236	NN_DIAG_NO_I_ DIDNOT	A response from a neighbour to an other chip, "no i have not heard from the neighbour on the link in Routing key bits 11-0". No payload is attached. It is a response to the sender's request.

237	NN_DIAG_ASK_N_TO_ RESET_LINKS	A message from a neighbour to an other chip, "ask the common neighbour to reset the link towards me and resend hello message while i am doing this on my side". No payload is attached.

A.4 Application Loading Flood-fill Process

The instructions used to control the application loading flood-fill process in the SpiNNaker computing system (as explained in Chapter 7) are listed in Table A.3:

Table A.3: Application Loading Floodfill Instructions.

Value	**Instruction**	**Meaning**
211	NN_FF_START	Start of block-level flood-fill. The block size given in routing key bits 11-0 and starting address in the payload.
212	NN_FF_END	End of current block-level broadcast message, block-level CRC in the payload (in case of standard 32 bit CRC).
213	NN_FF_INTERRUPT	Interrupt current broadcast.
214	NN_FF_RESUME	Resume the interrupted broadcast.
215	NN_FF_CRC	Block level checksum, size of CRC in the routing key bits 11-0, for 4-bytes of CRC in the payload (in case of CRC more than 32 bits).
216	NN_FF_REMAINING_CRC	Remaining 4-bytes of the block-level CRC (if CRC is more than 32 bits), sequence number in routing key bits 11-0.

Appendix B

SpiNNaker-Host PC Communication Protocol

B.1 Introduction

The SpiNNaker computing system is attached to a Host PC for system-wide configuration, application loading, fault-handing, and run-time user interaction with the system/application. This appendix describes the protocol devised to establish real-time communication between the Host PC and the SpiNNaker system, as described in Chapter 9. We have devised an instruction set to manage the Ethernet-based communication between the Host PC and the Host-connected CMP. A 4-byte instruction is passed along with 2-bytes of data in the Ethernet frame. A few instructions need further options, such as the one related to the flood-fill process or instructions specific to a particular CMP in the system. These options are passed in 12-16 bytes after the instruction. The frame format used in this communication is shown in Figure B. The frame uses a placeholder for the TCP/IP headers to support TCP/IP protocol on SpiNNaker, however, SpiNNaker-Host PC Communication Protocol is independent of the use of TCP/IP. The frame handler routine in the SpiNNaker boot-up code interprets the instructions directly from the frame. The TCP/IP headers are included to support applications relying on the TCP/IP protocol and to enable connecting SpiNNaker with TCP/IP network. In the absence of TCP/IP the Host PC uses TCP dump supported PCap library [Org09], that provides an access to the full Ethernet frame, to communicate with the SpiNNaker system using directly the

Ethernet frames. If SpiNNaker is directly connected to the Host PC Ethernet interface, the only required fields to establish the communication are the Ethernet address of the Host PC and that of the Ethernet Interface on the Host-connected chip. In that case, the TCP/IP header fields can be padded with some invalid data.

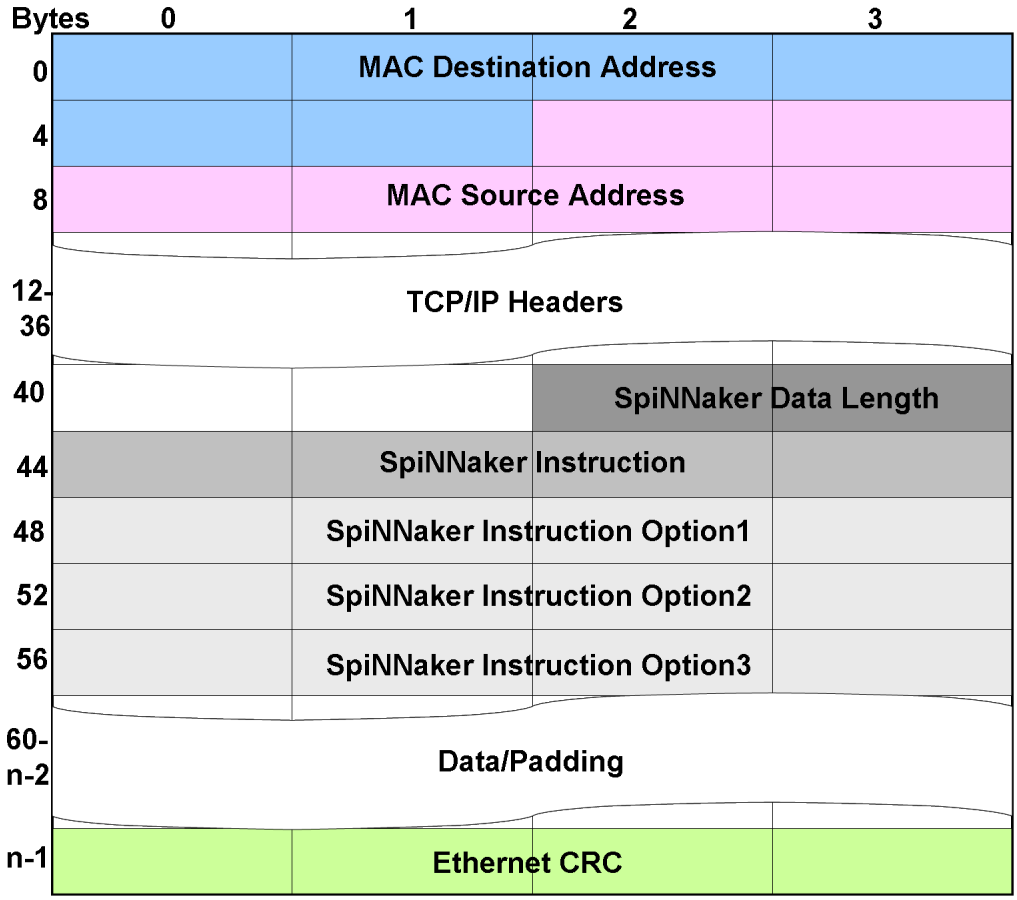

Figure B.1: The Ethernet Frame Format used for SpiNNaker-Host Communication.

The option fields after the SpiNNaker instruction are used for some specific instructions requiring more information on how to handle the data in the data field. During the flood-fill process some extra options are sent with the instruction in the option fields following the instruction in the Ethernet frame. The data-block size is sent in the 4 bytes of the SpiNNaker Instruction Option1 field, which is followed by the start address of the data-block in the next 4 bytes i.e.

the SpiNNaker Instruction Option2. The Host PC sends the block-level CRC for the data-block in the next 4-bytes i.e. in Instruction Option3 field.

The following sections list the instructions used for the Ethernet and P2P communication protocols.

B.2 Ethernet Frame Instructions

Some important instructions which are used as part of the SpiNNaker-Host communication protocol are listed in Table B.1:

Table B.1: Host-System Communication Instructions.

Value	**Instruction**	**Meaning**
51	WAIT_FOR_HELLO	Message from the Host PC to the Host-connected chip or from the connected chip to the Host PC, "please wait for my response, I am busy at the moment".
52	HELLO	Message from the Host PC to the Host-connected chip or from the connected chip to the Host PC, "hello I am ready to start the handshake".
53	FLOOD_FILL	Message from the Host PC to the Host-connected chip, "start block-level flood fill". The data block (1-K at the moment), is sent with the frame. The next word in the frame after the instruction bytes contains the size of the block. The second word after the instruction bytes contains the physical address to copy the block to, and the third word contains a 32-bit block-level CRC. The data block starts from the fourth word after the instruction.
54	END_FLOOD_FILL	Message from the Host PC to the Host-connected chip, "end of the current flood-fill process". The Host PC expects an ack frame.

55	SEND_ACK_FRAME	Message from the Host PC to the Host-connected chip, forcing the Host-connected chip to send an ack frame for which the Host PC has been waiting for a long time.
56	BREAK_SYMMETRY	Message from the Host PC to the Host-connected chip, "Host-connected chip to assign itself (0,0) address and then ask the other chips to compute and assign their relative chip addresses".
57	SEND_STATUS_REPORT	Message from the Host PC to the Host-connected chip, "accumulate and send the chip status for all the chips."
58	REPORT_OK	Message from the Host-connected chip to the Host PC, "the frame contains the status of the chips in pairs of two words" i.e. a word contains the chip address followed by its status in next 32 bits.
59	P2P_IN_COMM	Message from the Host PC to the Host-connected chip, "point to point communication for a particular chip", eg. sending routing table entries etc. The word after the instruction contains the destination chip's address, while the word next to this will contain the size of data for the destination chip. The data block to be sent to the destination chip starts from the fourth word after the instruction.
60	P2P_OUT_COMM	Message from the Host-connected chip to the Host PC, a response from the source chip to the Host PC. In case of the data to be sent to the Host PC, the Host-connected chip will accumulate the data to form a frame to be sent to the Host PC.

61	START_APPLICATION	Message from the Host PC to the Host-connected chip, "the application has been loaded, start executing it from address X". Address X in the word after the instruction.
62	WAIT_FOR_ACK	Message from the Host PC to the Host-connected chip or from the connected chip to the Host PC, "wait for ack as I am busy doing the previous job".

B.3 P2P Communication Instructions

We have devised a protocol to support run-time interactive communication between the Host PC and a particular chip in the multi-CMP SpiNNaker system. This communication is essential for the following purposes:

- Network Configuration: As part of system-wide configuration, we need to configure the on-chip routing tables on each SpiNNaker CMP to support the application to simulate a particular neural network. The routing tables are produced outside the system during application configuration process which computes the mapping for the neurons on processors on each CMP, the lookup tables for the synaptic data in the SDRAM, and the routing tables for each chip. This data is specific to each chip and can not be flood-filled.

- System-level Management: We need to periodically acquire the state of the system and the Communication Network to update the user interface. The system's state is obtained by interrogating the monitor processor on each chip to report the state of the chip and its communication activity. This requires communicating with individual chips or the chip may need to update their state periodically by sending P2P packets to the Host PC.

- Fault-handling: The faults, which can not be handled locally by the monitor processor, are reported to the Host PC. The Host PC then needs to communicate with the faulty chip or to its neighbouring chips to recover it. It is done with P2P communication between the Host PC and a particular chip in the system.

- Application Support: The application may require the user to interact with it at run-time for either to pass on stimuli/responses or to report a particular state. Besides this, the user may wish to visualize the state of any particular part of the neural network (neurons' state or synaptic data pertaining to some neurons) during the application execution. This requires P2P communication between the Host PC and a particular CMP.

For the reasons mentioned above, we have devised a protocol to support this communication using Ethernet frames and P2P packets. The communication between the Host PC and the Host-connected chip takes place using Ethernet Frames, whereas the Host-connected chip transforms this communication into a packet-based communication to pass the messages to a specific chip, or vice versa.

For the Ethernet communication we use the P2P instruction set given in Table B.2 to support P2P communication, in addition to the instructions given in Table B.1 to help the monitor processor interpret the Ethernet frame. The P2P instructions are passed in the first 2 bytes of the SpiNNaker Instruction Option1 field of the Ethernet frame as shown in Figure B.1. The address of the destination chip is passed in the remaining 2 bytes of SpiNNaker Instruction Option1 field. In the case of a data transfer, the size of the data is passed in the SpiNNaker Instruction Option2 field of the frame and a block-level CRC is passed as the SpiNNaker Instruction Option3.

On receipt of an Ethernet Frame for a P2P communication, the monitor processor on the Host-connected chip sends messages to the specific chips using P2P packets. To establish a reliable P2P communication over the SpiNNaker Communication Network, we have devised an acknowledgement based protocol. In case of a single instruction, the destination chip sends an acknowledgement after receiving the packet. While, for data transfer, we use a sliding window flow control protocol with window size 4 which is controlled using the "sequence number" field of the P2P packet (Chapter 4 Figure 4.5). As per this protocol, the destination chip sends acknowledgement packet on receiving '0b11' in the sequence number (or the end of communication if it happens before receiving a packet with '0b11'). The instructions are sent as part of the payload in the P2P packet containing 0x08F in its bits 31-20 to indicate that the payload contains an instruction. The instruction is sent in bits 19-12 and the size of data block to follow is sent in bits 11-0. The data packets will contain 32-bit data in their payload with a 2-bit incremental sequence number in the packet's control field.

The routing key contains a 16-bit destination address and a 16-bit source address for routing purposes. The block-level CRC is sent at the end of the data-block, followed by an instruction indicating an end to the transmission. The same 8-bit P2P instructions are used in both the Ethernet frame and P2P packets. Some of the P2P instructions are given in Table B.2:

Table B.2: Inter-CMP P2P Communication Instructions.

Value	**Instruction**	**Meaning**
21	P2P_COMM_REPORT_ STATUS	A message from the source chip to the destination chip, "send me the chip status". The sender expects a response. Response to piggy back the ack packet.
22	P2P_COMM_GET_DATA	A message from the source chip to the destination chip, "receive the data to be copied to memory", size of data in the bits 11-0 of the payload. Next packet to carry the starting address of the block to be copied. The sender expects an ack.
23	P2P_COMM_SEND_DATA	A message from the source chip to the destination chip, "send the data from the memory", size of data in the bits 11-0, next packet to carry the starting address of the block to be copied. The sender expects an ack followed by a series of packets to carry data.
24	P2P_COMM_DIAG_CHIP	A message from the source chip to the destination chip, "be the nurse chip to repair chip on your port X". Port number X is sent in bits 11-0 of the payload. Ack is expected by the sender.
25	P2P_COMM_RESET_CHIP	A message from the source chip to the destination chip, "reset chip on your port in bits 11-0 of the payload".

26	P2P_COMM_DISABLE_CHIP	A message from the source chip to the destination chip, "disable the processors on chip on your port number given in bits 11-0 of the payload".
27	P2P_COMM_DISABLE_NN_PROC	A message from the source chip to the destination chip, "disable the processor (processor ID in bits 11-8), on your neighbouring chip towards your port number given in bits 7-0 of the payload".
28	P2P_COMM_RESET_LINK	A message from the source chip to the destination chip, "reset your links corresponding to high-bits (1) in bits 11-0 of the payload".
29	P2P_COMM_DISABLE_LINK	A message from the source chip to the destination chip, "disable your link corresponding to high-bits (set to 1) in bits 11-0 of the payload".
30	P2P_COMM_DATA_FOR_HOST	A message from the source chip to the destination chip, "receive data as a response to your request request".
31	P2P_COMM_ACK	A message from the source chip to the destination chip, "ack for the P2P packet received".