SCALABLE EVENT-DRIVEN MODELLING ARCHITECTURES FOR NEUROMIMETIC HARDWARE

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

2011

By
Alexander D. Rast
School of Computer Science

Contents

Word Count: 62848

List of Figures

Abstract

The University of Manchester
Alexander Rast
For the degree of Doctor of Philosophy
Scalable Event-Driven Modelling Architectures for Neuromimetic Hardware
26 January, 2011

Neural networks present a fundamentally different model of computation from the conventional sequential digital model. Dedicated hardware may thus be more suitable for executing them. Given that there is no clear consensus on the model of computation in the brain, model flexibility is at least as important a characteristic of neural hardware as is performance acceleration. The SpiNNaker chip is an example of the emerging "neuromimetic" architecture, a universal platform that specialises the hardware for neural networks but allows flexibility in model choice. It integrates four key attributes: native parallelism, event-driven processing, incoherent memory and incremental reconfiguration, in a system combining an array of general-purpose processors with a configurable asynchronous interconnect.

Making such a device usable in practice requires an environment for instantiating neural models on the chip that allows the user to focus on model characteristics rather than on hardware details. The central part of this system is a library of predesigned, "drop-in" event-driven neural components that specify their specific implementation on SpiNNaker. Three exemplar models: two spiking networks and a multilayer perceptron network, illustrate techniques that provide a basis for the library and demonstrate a reference methodology that can be extended to support third-party library components not only on SpiNNaker but on any configurable neuromimetic platform. Experiments demonstrate the capability of the library model to implement efficient on-chip neural networks, but also reveal important hardware limitations, particularly with respect to communications, that require careful design.

The ultimate goal is the creation of a library-based development system that allows neural modellers to work in the high-level environment of their choice, using an automated tool chain to create the appropriate SpiNNaker instantiation. Such a system would enable the use of the hardware to explore abstractions of biological neurodynamics that underpin a functional model of neural computation.

Declaration

Copyright

Acknowledgements

This work could not have been carried out without the support of the EPSRC and ARM. I would like to thank Steve Furber for his insightful advice and brilliant suggestions - particularly in bringing to light the *principles* behind neural design. I would also like to thank Stephen Welbourne and Francesco Galluppi for their tireless efforts in model experimentation and the inspiration they brought from the psychological community. Two collaborators, Sergio Davies and Mukaram Khan, assisted repeatedly with low-level debugging and technical issues; without their help the models I built might never have worked in practice. Luis Plana gave critical debugging support at critical times - with a level of professionalism and skill I stand in awe of. David Lester made sure that my ideas remained grounded in reality - along with offering not a few good ideas of his own! All the members of the SpiNNaker team provided encouragement and assistance along the way - and demonstrate to the full the value of a collaborative effort. Finally, I give special thanks to Viv Woods who assisted countless times in practical matters and in navigating the complexities of “the system” - without whose help I might well have ended marooned in Hong Kong, Italy, or other points around the world, to say nothing of within the University!

Chapter 1

Introduction

1.1 Appropriate Neural Development Systems

Neural networks offer a different model of computation from conventional sequential digital computers, and one that achieves in biology what (thus far) technology has been unable to do: *intelligence*: the ability to compute unprompted and produce meaningful output. If there is a sense that figuring out the neural model of computation would make it possible to realise computational intelligence, as well as better understand how the brain works, there is also a recognition that neural science is at least as far away from being able to give a definitive answer to this as computer science is from creating an artificial intelligence. Vexing problems of model scaling and parameter proliferation make it difficult to decide what parts of the neural model of computation are critical, and formal models of structure and dynamics tend to exist only for narrow subdomains of neurocomputing. The easiest approach appears to be to come at the problem from the reverse direction: instead of trying to understand biological observations and use them to construct computers, build computational models and see what they can tell us about the biology.

The basic purpose of computational neural networks in this approach is to answer the question: *what does neural computation do?*. For many years, researchers have attempted to address this question using conventional computers. Yet given that neural networks present an emphatically different model of computation from the sequential digital model it is unclear whether running neural networks on industry-standard computer architectures represents a good, much less an optimum, implementation strategy. With further progress on understanding neural computing also appearing to depend on radical model scaling [IE08] dedicated neural hardware also looks more and more like a prerequisite [JL07], if only for the purposes of model acceleration. If in the past, however, the purpose of neural hardware was obvious: model acceleration; the nature of the research question for new chips has changed. Now it is not simply a matter of *how* to scale a neural model to large sizes but *what* model to scale.

The question of which neural network models are the "right" ones depends rather critically on what the purpose of the model is. A "one-size-fits-all" hardware neural model is thus unrealistic and somewhat irrelevant. Different research groups, with different goals, may want to use neural hardware in different ways, and it is unrealistic to expect them to have, or acquire, the low-level hardware familiarity necessary to develop efficient models for such chips. All of these considerations form part of the background for the subject of this work: development systems for neural hardware, a problem largely overlooked to date in neural research.

Current neural systems suffer from an accessibility gap: fixed hardware and nonstandard software tools make it difficult for a potential user to model anything beyond simple applications. But adapting industry-standard tools and hardware is usually not an option because in them the synchronous digital model of computation tends to be an assumption taken as a given. Effectively, the neural model design and development stages become the exclusive domain of system specialists. The neural systems accessibility gap is, in essence, a symptom of an architectural gap. An important and biologically relevant alternative exists: *event-driven* computing, using inputs themselves to drive updates rather than an independent clock. For event-driven neural computation to be a viable architecture, however, it must overcome the accessibility gap. To bridge that gap, this work develops tools, methodologies, and libraries for general-purpose event-driven neural systems that a modeller can use to implement an arbitrary model.

1.1.1 Effective Tools for Model Exploration

At the level of the neural model, two important considerations emerge that an architecture for neural modelling libraries must address: universality - the ability to describe and run any model, and abstraction - the ability to describe a model independently of hardware or biological detail. Lack of clear consensus on what constitutes an adequate neural model has led to a tremendous diversity of different models, their characteristics determined as much by research priorities as by agreement with observation. On the one hand, this places a high value on neural development systems that can support any model - so that it provides the maximum scope for model exploration. On the other, it emphasizes the need for abstraction - so that it is possible to compare different neural models at a high level. Yet in practice existing neural hardware has tended to support only a small model subset, with correspondingly specialised development tools, and thus the concept of universal modelling libraries for hardware has been largely overlooked. With a new class of neural hardware emerging, however, that supports a wide variety of models on a dedicated, yet flexible hardware platform, this study reexamines the architecture of neural modelling systems, to create a system that can meet the needs of universality and abstraction on specialised neural hardware.

1.1.2 Effective Tools for Hardware Implementation

The existence of universal neural hardware has begun to address a need for systems for large-scale neural model exploration - but is only half of the *system*. To be effective such chips need to be part of an integrated *system* architecture, with hardware and tools designed together for universal neural modelling [Rey03]. In particular, a bare chip needs a development system that can specify and configure what neural model the chip runs. To be useful to the end user, this system should provide enough hardware abstraction that the modeller can describe and run the neural network entirely at the level of the model, without having to consider the hardware specifications. At the same time it must make optimal use of hardware resources, or it becomes questionable whether the system as a whole is a dedicated neural modelling platform. In emerging neural hardware the event-driven computing model is rapidly becoming an architectural standard. The event-driven model provides a useful generalised framework for abstraction, both from the point of view of isolating hardware details from the modeller and from the point of view of decoupling neural models from biophysical dependencies. To use this model in the configuration tool chain requires a method for recasting existing functional models into an event-driven format. This study considers an "ideal" architecture for the neural hardware development environment under consideration, taking the form of an event-driven system with a clearly-defined abstract computing model and a method to convert an arbitrary neural network to this model.

1.2 The Integral Role of the Development System

1.2.1 The Debate Over the Biological Model

While biological investigations have revealed much, particularly recently, about low-level neural processes, there is as yet scanty information and little consensus on higher-level neural function and how it produces behaviour [MSK98]. The result has been a proliferation of models, each with different and possibly conflicting objectives and claims to biological realism [Izh04]. Solving large-scale behavioural questions with biological accuracy requires impractical simulation times on conventional computers [IE08], [HMS08]; it seems virtually self-evident that dedicated neural hardware is essential. Neural network chips have been around for about 20 years now, however, the biological community has not received chips that attempt a literal hardwiring of a specific neural model in silicon enthusiastically, for two important reasons. Obviously, if the "correct" model of biological neural network operation is still under debate, hardwiring model assumptions into the design makes blind compromises while limiting the group of modellers who can use the device at all. Secondly, many of the model approximations in the hardware do not have enough theoretical support to be justifiable to biologists, whose models often involve very careful curve fitting. But a neural network system that attempts to learn by imitation, precisely replicating every detail of the biology or every line of an algorithm is unlikely to yield any fundamental insights. It is simply an implementation of what the researchers already know. If their utility is to extend beyond verification of pre-existing models, neural systems need a mechanism for abstraction.

To address these problems, a new device that implements a type of neural "blank slate" has been introduced: SpiNNaker, a chip with a neural network architecture but general-purpose components that can, in principle, simulate *any* model. However, by suggesting a form for the appropriate *hardware* architecture for neural modelling, SpiNNaker reopens the question of the appropriate *system* architecture: how are modellers to implement a given neural model on such a blank-slate device? In a system where the nature of its function depends on a model it has been configured to run, the tool architecture has a large part in determining the system's capabilities. Without a whole-system architectural approach the chip would be no different from earlier hardwired designs in that it essentially presented a chip to modellers and expected them to use it. This work addresses the critical need to develop an exemplar flow for neural implementation on universal hardware such as SpiNNaker, valid for multiple heterogeneous network models.

1.2.2 The Search for the Computational Model

Regardless of any biological insights, neural networks as a computational tool have proven to be a useful alternative to the conventional sequential digital computer for many practical problems. Given a bare neural chip, it is quite likely that individual developers would develop applications tightly optimised for the specific problem they were trying to solve, giving little scope for interoperability or design reuse. This reason alone is sufficient to justify libraries for neural instantiation on chips such as SpiNNaker. However, a more fundamental reason is that, just as the asynchronous parallel nature of biology can contribute insights to computation, the universal abstraction power of computational techniques may suggest better models for cognitive neuroscience. There remains a gap between observations and applications that needs improved theoretical support to narrow. Neural network design, even for simple problems, remains today very much a process of trial and error; extensive parameter tuning is the normal order of the day [GAS08]. Meanwhile, much of current neural computation theory consists of highly abstract proofs of formal properties. There are in contrast very few practical, concrete "design rules" that permit a modeller to *calculate* parameters or connectivity. This study introduces a library-based development model for neural hardware to permit like-for-like comparisons at a high level, allowing direct verification by experiment of principles which up to now could only be elaborated on paper.

The event-driven model is an important process abstraction with direct biological relevance as well as interesting computational properties. Real biological neural networks use spikes, and it is natural to abstract a spike to an event (assuming the precise spike shape is unimportant). There is some theoretical evidence to suggest that spiking neural networks have greater computational power than nonspiking systems [ŠO03], indeed, it is possible in biology that spiking systems with asynchronous dynamics are *necessary* to certain behaviours [THA$^+$10]. This makes the event-driven model the process dynamic of choice for a neural system architecture. Neural hardware such as SpiNNaker is already adopting event-driven communications as a standard, and thus the development systems for such hardware must likewise follow an event-driven model to maintain a unified system architecture. Since many classical computational neural models use continuous, nonspiking activations, the tool chain also needs to provide a method of translating nonspiking models into an event-driven representation. Event-driven development is nontrivial, however, and event-driven tools are scarce for any platform, much less a dedicated neural device. This work therefore develops *event-driven* libraries that can automate the process of instantiating a neural model: a basic requirement for a configurable neural system whose purpose is to uncover the principles of neural computation.

1.2.3 The Limitations of the Simulation Model

Without an integrated tool chain, bare hardware by itself is useless: it is cumbersome to run, much less design, models. Furthermore the typical neural modeller has little interest in or possibly even opportunity to learn a nonstandard, hardware-specific software system. Classical programming languages like C or Java do not intrinsically incorporate events and so are likely to fit poorly with the neural model; existing neural simulators like NEURON or GENESIS, or even Matlab, offer no easy extensibility to hardware and are unrealistic for real-world environments. Hardware design languages include native event support that fits elegantly with the chip and the neural model, but are unfamiliar outside a specialised audience. An approach that hides a hardware-design "back end" behind a neural-modelling "front end" is the type of integrated tool chain necessary. In this model the back-end contains prewritten, probably precompiled, "neural function" libraries that standard neural simulators can instance and use as if the hardware were part of the simulator itself. Thus the libraries provide the event-driven support and the neural simulator can deal purely in model-level abstractions. Note also that such an approach would be impossible without a *universal* neural device; "hardwired" chips would force the simulator to conform to its internal configuration, prohibiting model-level abstraction. To demonstrate the viability of this neural modelling architecture, this study demonstrates the core of the back-end libraries: efficient, non-trivial event-driven model implementations on the target hardware, and methods to integrate these into existing neural simulators, prototypes of the front-end interface. It will become clear through actual neural network implementations that an integrated, event-driven, library-based hardware system permits more efficient modelling of large-scale neural networks than either software simulators or bare custom hardware alone.

1.3 Contributions

1.3.1 Achievements

Development of an integrated modelling architecture for neural hardware establishes the value of **an event-driven library for universal neural hardware**. Within the framework of this overarching achievement, the following key contributions stand out as significant advances.

A system architecture for mapping a neural network to dedicated hardware:

If a universal dedicated neural device like SpiNNaker represents the most promising approach to large-scale modelling in light of debates over the model, it is also virtually useless without an integrated development system to instantiate models on the hardware. This work develops that model. It emphasises a view that considers the entire hardware/software platform as a unified *system*, and creates an architecture suitable for implementing neural networks not only on SpiNNaker but on other hardware platforms.

A general-purpose event-driven representation for neural models:

The neural library which represents the core of this work includes multiple heterogeneous models. Managing such models would be difficult without a common specification that makes it possible to build general interfaces into which the model details can simply be "plugged in". An important contribution is the development of a function pipeline, suitable for broad classes of neural model, and designed around considerations of what event-driven hardware like SpiNNaker can efficiently implement, that forms a "building-block" specification for neural models.

Scalable implementation of neural models in event-driven hardware:

Especially when large-scale networks containing more than $\sim 65K$ neurons are under consideration, design automation is critical. In addition to the software architecture and libraries, this work develops a functional automated tool chain that allows a user to specify a model in a high-level description environment and have the tools automatically configure, map, and implement the model on SpiNNaker hardware. Implementation of functional neural networks able to operate in real-world conditions demonstrates the utility of this architecture and provides an exemplar template for future large-scale model implementations.

Important details on these contributions have appeared in the following:

1. 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), pp. 2727-2734, 2008.

2. A.D. Rast, X. Jin, M. Khan, and S. Furber, “The Deferred-Event Model for Hardware-Oriented Spiking Neural Networks”. In Proc. 2008 Int’l Conf. Neural Information Processing (ICONIP 2008). pp. 1057-1064, 2009.

3. 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), pp. 2611-2618, 2009.

4. A.D. Rast, S. Welbourne, X. Jin, and S.B. Furber, “Optimal Connectivity in Hardware-Targetted MLP Networks”. In Proc. 2009 Int’l Joint Conf. on Neural Networks (IJCNN2009), pp. 2619-2626, 2009.

5. A.D. Rast, X. Jin, F. Galluppi, C. Patterson, M.M. Khan, L.A. Plana, and S.B. Furber, “Scalable Event-Driven Native Parallel Processing: the SpiNNaker Neuromimetic System” In Proc. 2010 ACM Int’l Conf. Computing Frontiers, pp. 21-29, 2010

6. A.D. Rast, F. Galluppi, X. Jin, and S.B. Furber, “The Leaky Integrate-and-Fire Model: A Platform for Synaptic Model Exploration on the SpiNNaker Chip” In Proc. 2010 Int’l Joint Conf. Neural Networks, pp. 3959-3966, 2010

7. 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), pp. 2849-2856, 2008.

8. X. Jin, A. Rast, F. Galluppi, M. M. Khan, and S. Furber. “Implementing Learning on the SpiNNaker Universal Neural Chip Multiprocessor”, In Proc. 2009 Int’l Conf. Neural Information Processing (ICONIP 2009). Springer-Verlag, 2009.

9. 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 Proc. Int'l. Symp. on Parallel and Distributed Computing (ISPDC2009), pp. 54-61, 2009.

10. X. Jin, M. Lujan, M.M. Khan, L.A. Plana, A.D. Rast, S. Welbourne, and S.B. Furber, "Efficient Parallel Implementation of a Multi-Layer Backpropagation Network on Torus-connected CMPs" In Proc. 2010 ACM Int'l Conf. Computing Frontiers, pp. 89-90, 2010

11. X. Jin, M. Lujan, M. Khan, L.A. Plana, A. Rast, S. Welbourne and Steve Furber "Algorithm for Mapping Multilayer BP Networks onto the SpiNNaker Neuromorphic Hardware" In Proc. 9th Int'l Symp. Parallel and Distributed Computing (ISPDC 2010), pp. 9-16, 2010

12. X. Jin, A. Rast, S. Davies, F. Galluppi, and S. Furber. "Implementing Spike-Timing Dependent Plasticity on SpiNNaker", In Proc. 2010 Int'l Joint Conf. Neural Networks, pp. 2302-2309, 2010

13. X. Jin, F. Galluppi, C. Patterson, A. Rast, S. Davies, S. Temple, and S. Furber. "Algorithm and Software for Simulation of Spiking Neural Networks on the Multi-Chip SpiNNaker System", In Proc. 2010 Int'l Joint Conf. Neural Networks (IJCNN 2010), pp. 649-656, 2010

1.3.2 Outline

The rest of the work will develop the themes and contributions this Introduction has outlined briefly, through the following sections:

Review of Hardware-Based Neural Systems

This chapter will discuss the historical context of neural modelling on hardware. It will outline previous important architectures and findings, and give some feeling for the overall direction of research trends through time.

Introduction to the SpiNNaker Chip

SpiNNaker is the chosen hardware platform to demonstrate event-driven models, and the exemplar universal neural device. It is essential in order to understand the actual model development to have a working knowledge of the hardware design and features of SpiNNaker. This chapter will discuss SpiNNaker in overview.

The Library Architecture

This chapter introduces the core architectural concepts that form the central subject of the research. It goes through the process of selecting and defining the design tools and the development model for SpiNNaker. Such a process is not trivial because most tools incorporate an implicit synchronous system assumption. The question is how to implement a system that allows the user to create event-driven models without detailed hardware knowledge.

Network Implementation

This chapter describes the actual networks implemented on the SpiNNaker platform. It will describe their structural and dynamic design to make test results intelligible, but will focus on presenting the networks in the general context of an event-driven function pipeline. The emphasis of this chapter, therefore, will be the architectural model rather than the details of any given network.

Software Testing: Pre-Hardware Implementation

This chapter will report results from testing in simulation. Most of the development work used Verilog and SystemC simulators to present a virtual chip test environment, and this chapter both presents the simulation results and indicates the limitations of software simulation. Finally, it will examine the importance of software testing as part of an overall neural system design process.

Hardware Testing: On-Chip Implementation

This chapter will report results from testing on physical SpiNNaker hardware. Real hardware permits testing of much larger-scale networks, thus the chapter will focus on large-scale tests. It will only note smaller scale tests where there is a significant observed deviation from software testing.

Discussion: Overall Findings

This chapter will present the implications of the event-driven model: what it reveals about necessary models of computation for neural systems. It will also discuss what the benefits of such a model are: what new computing capabilities become available. Finally, it will try to give some perspective on lessons learned: what building and testing revealed about the assumptions built into current models and design tools, which implementation tactics work and which turn out to be impractical.

Architectures for the Present and the Future

This chapter will offer some conclusions. The presentation will reflect a strong opinion that real conclusions consist not in extracting the immediate implications of the research (this is the role of a discussion section), but on what influences the research might have on the field as a whole. In this case, the central point is that neural networks introduce an alternative model of computation that will mean entirely new system architectures, and that it is through this architectural reconceptualisation that it will become possible to understand how the brain works.

1.3.3 Significance

What is the impact of event-driven libraries for neural hardware? If there is one central notion that this thesis wishes to show it is that **neural computation is a fundamentally different model of computing than sequential digital computation.** To that end, this research makes and will attempt to support the following central claims.

Neural networks need universal dedicated hardware.

A neural network simulation running on a general-purpose serial computer is two virtually independent computational models trying to run at the same time. This is inefficient, impractical for very large networks, and eliminates one of the central benefits of neural computing: parallelism. Indeed, it is not even clear whether a serial, synchronous system can model all the dynamics of a parallel network, some of which may depend explicitly on the architecture. Therefore, hardware that offers a native implementation of the neural computing model is essential.

Useful neural modelling requires abstraction of space, time, and function.

As long as the debate exists over the precise nature of neural computation, no design that fixes the model in silicon can be authoritative. However, a “silicon copy” of the brain provides no insight either, for replication of known neurobiology is simply empirical repetition. A neural network model must therefore be able to provide an abstraction that successfully predicts neural behaviour if it is to increase understanding. Since neither the internal function, topology or temporal dynamics of neural networks are fully understood, the model must provide a mechanism to abstract each.

Event-driven neural computation provides a general means for temporal abstraction.

No system that depends upon an external clock can provide a complete abstraction of time, because the external clock introduces an absolute reference. Synchronous sequential computation is thus inadequate and inappropriate for neural networks. The event-driven model introduces a new and fundamentally different model of time: event rate. Events can carry arbitrary time labels (including no time marker), thus it is possible to implement any model of time simply by specifying the label for each event to carry.

An integrated, library-based tool set is a core architectural component of a universal neural system.

If the critical feature of universal neural hardware is user configurability, it must needs have development tools that make it possible for the user to configure the chip. Tools that only provide low-level, assembly-language-like programs consign the hardware to a small audience because very few users are willing to negotiate the steep learning curve and long development cycle necessary at such a low level. Libraries are an easy way to provide this high-level functionality without losing implementation efficiency. If the tool set contains prewritten, low-level libraries of neural function, that the user can instantiate via a script-based system or GUI at a high level, a new user can quickly create and run neural networks on the hardware.

A function pipeline containing retrieval, evaluation, lookup table, interpolation, and differential equation solution is a block-level abstraction sufficient to implement an arbitrary neural model.

If different neural model implementations use a completely different module architecture, library management becomes too complex and interoperability in the same simulation nearly impossible. A system based on block-level replication of standardised function templates makes it easy to create the high-level structure while retaining model flexibility through parameterisation. Such a system, furthermore, resembles the architecture of the brain and this close correspondence to an evolved solution suggests that it will be an efficient method with a minimum of implementation complexities. Considering what circuit fragments can implement what operations efficiently, and what operations neural networks require, the general block template should have variable retrieval, polynomial evaluation, table lookup, polynomial interpolation, and differential equation solution as its subcomponents.

Abstracted models running on generalised hardware can reveal enough biological insight to be meaningful in understanding the brain.

There is no reason to believe a working abstract model running on abstract hardware inherently has any claims to biological plausibility. This makes it difficult to justify the validity of the model, especially in the eyes of neurobiologists. On the other hand, if the biological model of computing depended on all the biophysical minutiæ, neural computation would be a sterile research field: there would be no way to derive theoretical principles of neurocomputing, leaving it a purely empirical discipline. Therefore if neural computing research is to be fruitful it *must* be the case that abstract models can replicate the essential functionality of the brain. A model running on universal neural hardware that duplicates important behavioural characteristics and has strong predictive power with respect to other, previously uncorrelated behaviours, is powerful evidence for the value to biology of the event-driven architecture and and the universal neural chip.

A universal event-driven system architecture is the critical abstraction necessary to achieve brain-like levels of neural modelling and computing capability.

Attempts to emulate full-brain function on synchronous digital systems run into increasingly intractable scaling barriers. On the other hand, attempts to duplicate neural behaviour even on small scales with increasing levels of biophysical fidelity run into increasing intractable analysis barriers. When computation is completely deterministic it is, indeed, questionable whether *any* model can fully explain or achieve brain-like computation. Changing the model to an event-driven architecture relieves scaling difficulties while providing a level of non-determinism that can sustain spontaneous activity. Combining the event-driven model with a universal neural hardware system contains the analytical explosion while moving towards a native neural execution platform rather than a force-fit adaptation of digital systems.

Event-driven neural computation presents a meaningful alternative to sequential digital computation.

Progress in computing will continue to stagnate so long as the serial uniprocessor computing model is taken as an unquestioned given - the situation that prevails today. Clearly, neural networks are not serial uniprocessors: attempts to fit them to this model will fail. Ultimately, the brain solves a different class of computing problem from the digital computer. Solving this class of problem may well require brain-like neural computing. Dedicated, nondeterministic parallel systems thus become a basic requirement for neural computation, at least at large scales; it is the event-driven architecture that provides a usable execution model for such systems.

Chapter 2

Neural Network Architecture Trends

2.1 Context

This chapter will attempt to put neural network hardware in historical context and review the main research developments. The first part considers the architectures: basic platforms for neural networks that have historically dictated available capabilities. These architectures fall into three basic types, “hardwired” networks which have attempted a direct hardware implementation of a specific neural network algorithm, FPGA-type networks which offer reconfigurability and wider model choice in exchange for somewhat reduced speed and scalability, and neuromorphic networks, which move beyond simplified computational algorithms to attempt biorealistic modelling, often with some level of configurability. A pervasive theme permeating all these approaches has been “direct implementation” - the alluring but ultimately somewhat misleading concept that it should be possible to make the hardware a literal embodiment of the model of computation.

The second part asks a critical question that affects the value of a “direct” implementation: what is the hardware going to be used for? There is no reason to believe, *a priori*, that hardware designed for biological simulation will perform well in computational applications, and it is important within the biological domain to consider what part of the neural network is most important within the context of a given study: network dynamics, neural behaviour, synaptic response. In the computational domain, meanwhile, neural networks perform well for *specific* applications but are not necessarily good for general-purpose computation. One domain where neural networks may be particularly valuable, however, is embedded systems. Such applications usually demand real-time adaptability to dynamic environments, often under tight power constraints. Standard digital processing is frequently unsuitable in such situations and here neural networks can have compelling advantages. Event-driven computing emerges as a particularly suitable model for the types of applications that neural hardware performs well in, with clear biological relevance, offering a path to standardisation which unifies the disparate applications for neural networks into a single architectural model.

The third part examines actual concrete implementations in hardware. Specifically, it looks at chips: while there have been some neural network hardware platforms that make original use of standard hardware, in the main the development has been in custom IC’s or FPGA’s. The story of chip development follows two major routes: analogue hardware which hopes to emulate neurons with silicon device physics, and digital systems, which hope to leverage the advantages of standard and fast-improving process technology along with vastly more predictable device characteristics in order to create accessible neural hardware that minimises the need for careful parameter tuning. However, the signal development in the field has been the introduction and rapid adoption of Address-Event Representation (AER), a mixed-signal communications standard that permits easy interfacing between heterogeneous components, transforming the analogue/digital choice from a hard commitment into an implementation convenience. AER is perhaps the most important advance in neural hardware systems and paves the way for "neuromimetic" hardware: chips with an abstract, configurable internal structure but an overall architecture optimised for neural processing.

2.2 Architectures: Hardwired, FPGA, Neuromorphic

A neural network is a structural architecture that naturally suggests hardware implementation. Dedicated hardware can fully replicate the parallelism of the network, and neural networks provide a classic example of an "embarrassingly parallel" application [NS92]. Unsurprisingly, therefore, neural network chips emerged almost as soon as the process technology reached the point that this was feasible [TH86], [GHJdV87], [Gar87]. Since the first neural network chips, however, process technology has evolved, and so has research opinion on the actual biological operation of neural networks [MJH$^+$03]. As a result there has been a comparable evolution in the prevailing hardware neural network architectures over time [RTB$^+$07]. The research community has had 3 main phases of development. In the first, the "hardwired" phase, the emphasis was on pure connectionist functionality in silicon, whether analogue or digital [LL95]. Devices in this phase also tended to use non-dynamic "timeless" models employing either continuous-time or synchronous update behaviour. In the second, "reconfigurable" phase, designs moved onto reprogrammable hardware, mostly digital FPGA's, either to time-multiplex functions or to permit dynamic network remapping [JS03]. At this point event-driven implementations begin to emerge, primarily for efficiency rather than biological realism. In the third, "neuromorphic" phase (still currently prevalent), focus shifts to greater biological plausibility with spiking networks, often analogue, frequently with extensive reconfigurability [JC09]. During this evolution, process technology tends to determine progress to a greater extent than biological (or even computational) research findings.

The dynamic model: how the neural network "state" evolves with time, has been an important strand of development. From the outset it has been clear that conventional sequential processing is a poor fit for modelling neural systems. [JSR+97] It is not clear that the model of a single master clock to drive internal state updates is optimal for parallel systems [YB09], and it certainly is not biologically plausible [TMS93]. This has given scope for researchers to experiment with various dynamic models, using neural processing as a "silicon canvas" to try their ideas [PWKR02b], [MJH+03], [ICD06]. In spite of this freedom, however, one major timing assumption continues to limit most current designs: the need to maintain a coherent system state. This is a stipulation that at any point the modeller can stop the chip in mid-process and get a complete "system snapshot" of the entire neural state [PLGW07]. Such an assumption distorts the model of parallel computation. A truly parallel computer is a set of independent processing modules; if system state must be coherent then the computer is not actually parallel but a set of coupled subprocesses in an overall serial architecture [CJ92], [MR94]. Historically, this is how early models of neural computation emerged [Ros58] [Hop82] but the result has been to create a dominant architectural model in the chip design community that persisted for some time after the neural modellers had already moved on to dynamic, spiking models. Not until FPGA implementations started to examine creative solutions to resource limitations [LSLG94] did researchers begin systematically to examine the temporal model, and it has only been relatively recently that neuromorphic systems have started to implement spiking [APR+96] as the standard model [Maa01]. There remain still strong traces of the coherent architecture even in state-of-the-art devices, however, and on the whole it appears that observability has had a controlling influence on the evolution of the dynamic model in hardware.

2.2.1 Hardwired Architectures

Early neural network chips were mostly hardwired [GHJdV87], [FWA88], [YMY+90]. In the hardwired approach, hardware, usually custom ASICs, directly implements a specific neural model and topology. Only minimal configuration of the chip is possible, if the device allows any programmability at all [HTCB89]. Hardwired designs achieve the highest speeds possible for the available process technology, since the hardware is a literal translation of the model into circuitry. While this architecture is the "obvious" approach it has significant drawbacks. There has tended to be no tool chain other than a basic interface, usually in the form of a device driver [SBS05]. With minimal tool support, using such chips has, historically, been difficult [ICK96]. The would-be modeller usually has to have intimate knowledge of the low-level details of the hardware. Integrating these chips, containing proprietary interfaces, into larger systems is equally difficult because the entire system must be built from the ground up [Ker92]. But the critical limitation of such chips is the obvious one: they can only model one, or at most a few, neural models. Therefore there is minimal scope for model experimentation, and unless the implemented model exactly matches the experimenter's preferred neural model, the device is in effect useless [SBS05]. Chips implementing true dynamic models are rare [SHS95]; those with any hardware support for events, rarer still, The effect has been to confine the use of these chips to the original developers, along with, at most, a small group of related researchers sympathetic to the model implemented. Given the very high up-front investment required to produce a full-custom ASIC, it has been very difficult to justify producing such devices, and hardwired architectures ended up being something of a research dead-end [Omo00].

2.2.2 Reconfigurable Architectures

Reconfigurable architectures use FPGA devices to implement neural networks with dynamic internal logic [EH94b]. FPGA implementations became popular when it became clear that the ASIC approach was gaining little adoption and when coincidentally FPGA densities started to become large enough that implementing full-scale neural networks on them was practical [PWKR02a]. They use commercially available, off-the-shelf hardware and development tools, making hardware neural modelling accessible, at least in principle, to those without the resources to develop a full-custom IC. However, FPGA's bring with them several significant limitations. Dynamic reconfigurability is only possible with some FPGA's, and furthermore the tools usually limit the scope for reconfiguration [SVS$^+$09]. Such reconfiguration takes time and is very complex to program, FPGA programming sequences being typically arcane and proprietary [KdlTRJ06]. FPGA's have digital components with a very specific structure, placing further limits on what functions and dynamics are implementable, and often requiring careful optimisation, particularly with expensive synaptic multiplication operations [HAM07]. Most importantly, FPGA's impose a clocked synchronous model of computation: any neural model that uses other models of time must transform or adapt its dynamics into the synchronous environment [WL06], [ROA$^+$06].

There have been 2 types of reconfigurable architecture: component-multiplexing and programmable-model. Component-multiplexing architectures use the reconfigurability to implement a much greater number of individual neural elements than would be possible with fixed hardware of the same size [EH94a]. The device multiplexes individual components, either performing different processing stages or implementing different neurons [GMM$^+$05], while in operation [LSLG94], [BHS98]. The component-multiplexing approach was the first to hint at an event-driven architecture, particularly with bit-stream applications that suggest an atomic message: an event, per bit [GSM94]. Using an event-driven approach to drive the multiplexing itself, however, remains a future research area [GMM$^+$05]. In part this is because the multiplexing itself incurs a time overhead, making it suitable mostly in coherent update applications where scheduling the multiplexing in the presence of asynchronous input does not present problems [HAM07]. For similar reasons the actual neural model implemented on chip is usually static, thus they do not exploit the intrinsic programmability of the FPGA.

Programmable-model architectures use the dynamic reconfiguration capabilities to change the actual topology of the neural model, or its internal dynamics [PWKR02b]. Many of the early implementations, given the known limitations of partial reconfigurablity, focussed on changing the topology [PUS96] [UPRS05]. A severe limitation with FPGA's is the circuit-switched architecture, which hampers the ability to create the dense connectivity patterns realistic neural networks require [HMH$^+$08b]. Attempts to create reconfigurable or event-driven dynamics do not appear in most cases to have proceeded beyond the design-exploration phase [EKR06], [ROA$^+$06]. This seems, in fact, to be somewhat of a summary of the development of neural networks on FPGA's: useful for design exploration, but not pursued for full-scale modelling [HMH$^+$08b].

Recently there has been experimentation with programmable-model architectures using graphics processors (GPU's) [OJ04]. The attractiveness of GPU's comes from massive parallelism with standard parts at minimal cost. Because of their particular design, it is also easier to incorporate reconfigurable dynamics on GPU's than FPGA's. However, the nature of the GPU: an application-specific processor optimised for a *different* application than neural networks, imposes FPGA-like architectural limitations [BK06]. GPU's use an emphatically synchronous, coherent model. Maintaining process coherency, in fact, turns out to be one of the most challenging research topics [NDKN07]. Various similar limitations mean that mapping a neural network to a GPU is nontrivial, limiting its use in practical terms to people with extensive hardware expertise [BK06].

Ultimately the critical limitation for both platforms is probably limited scalability. FPGA's use a circuit-switched architecture, and this in turn means wire density limits achievable connectivity [HMH$^+$08b]. It is not difficult to note that while model connectivity scales potentially as N^2 (where N is the number of neurons), FPGA wire density scales linearly with area (hence with number of neurons) and quickly runs out of routing resources with large networks. With GPU's, the external interfaces do not permit easy scalability to multichip systems [FRSL09]; not particularly surprising given that the design of the GPU assumes a single chip sitting on a graphics card. FPGA's are also very power-hungry: a design with 60% resource utilisation might consume 50W on a large FPGA like the Virtex-5. Meanwhile, GPU power requirements are crippling: an nVidia GeForce GTX280 can consume 236 W. Although reconfigurable architectures have proven useful for small-scale modelling and proof-of-concept experimentation, they are inadequate for large-scale neural networks.

2.2.3 Neuromorphic Architectures

Neuromorphic architectures are more intimately concerned with biological plausibility than previous designs, usually implementing spiking networks with full dynamics [PRAP97], [WNE99]. Relative to the FPGA, neuromorphic systems appear like a return to older hardwired architectures, possessing considerably less general-purpose reconfigurability than FPGA designs, but this reflects an integration of the knowledge gained in the previous generations [VMC$^+$07]. If the era of hardwired architectures revealed that fixing the model in hardware was overly limiting, the reconfigurable era showed that general-purpose reconfigurability sacrificed too many efficiencies in the name of universal application [HMH$^+$08a]. In particular, the FPGA design leads to a synchronous digital model of computation that is too power-hungry and area-intensive for elegant neural implementations [NdSMdS09]. Thus the neuromorphic approach accepts that some application-specific components are necessary, possibly using combinations of analogue and digital techniques [IWK99], along with enough general-purpose reconfigurability to tailor the system for various models [VMVC07]. There has therefore been a subtle shift of emphasis from which model to implement to what level of reconfigurability is appropriate to achieve an optimally efficient modelling platform.

Equally importantly, neuromorphic architectures incorporate the time domain integrally, in contrast to "timeless" hardwired architectures [WCIS07]. Therefore they can model biological networks with much greater realism than previous designs [VMVC07]. Typically, however, the networks integrate time explicitly in the hardware, as a parameter in analogue circuitry which usually tries to approximate the actual biophysics of neurons [MS08], [YC09]. Time is therefore fixed, with some scaling factor relative to "real" time, not abstract or programmable [KBSM09]. Where there are digital components they often (and increasingly, typically) use events [CWL$^+$07], but only as an implementation convenience: events are merely the unit of translation between analogue and digital domains. Usually such systems also preserve time-coherency, in the sense that it is possible to interrogate the digital interface and get a complete readout of the network state [PVLBC$^+$06], [CWL$^+$07]: this represents a major departure from biological realism where the notion of a global state is meaningless because no one local unit or region possesses an instantaneous global view [Ban01]: neural transmission involves real delays [CEVB97], [Izh06]. While undoubtedly the neuromorphic approach allows the closest hardware approximation to the biology thus far, limited ability to abstract time still means that the fundamental question modellers ask is how accurately the hardware models the biology rather than what models they could use with the hardware to find out more about the biology [BI07].

This problem deflects attention away from the central question: if much of the actual computational model of the brain remains unknown, how can hardware provide insights into its function by modelling the biology realistically [VMVC07]? In this regard current neuromorphic architectures suffer from several deficiencies. One is in the area of model support: existing implementations have tended to emphasize hardware efficiency at the expense of reconfigurability, bringing them closer to the hardwired chips with all their known limitations [KGH09]. The other major gap is tool support: neuromorphic architectures tend not to integrate seamlessly into existing simulators [BMD+09]. The user cannot therefore simply run a predeveloped model on the hardware: it needs translation, assuming, furthermore, that the target model is compatible with what the hardware supports [BGM+07]. Being outside mainstream commercial technology, such translation tools are scanty to nonexistent [BMD+09]. When it is difficult, such as here, to get even standard reference models to run, an understandable reaction of the biological modeller is to express scepticism about the validity of the models the chip implements [CSF04] - because he cannot test it against a model he is familiar with and trusts [RD07]. Thus neuromorphic chips have not to date provided a compelling platform for discovering valid biological abstractions, because it has been difficult to make direct comparisons against detailed models. The "ideal" architecture would be more of an integrated software/hardware system [OWLD05], containing a device that combined general-purpose programmability with the biological realism of current neuromorphic architectures, and a tool chain with direct support for existing simulators, native parallelism, and the ability to present neural networks at different levels of hardware abstraction.

2.3 Applications: Computational, Biological, Embedded

Whatever the potential benefits of a given neural architecture are, they have little meaning outside the context of an application in which to run. Neural network architecture is much more application-specific than digital computing, where the microprocessor realises a reasonably good solution to any problem that can be posed as a sequential algorithm, with an architecture that has stabilised to a standard form [Kön97]. Some of this may be intrinsic: it is possible that the physical form of the neural network model is tightly coupled to the task it performs [STE00], [ST01]. Evidence from neurobiological studies, however, indicate that such "hard-wiring" is limited and important only for primitive behavioural response or early processing stages; indeed, it may not even remain active after development [CHC06]. Much of the observed tailoring of architecture to application in the computational domain occurs rather because current applications come from fields with completely different priorities. There are applications in "classical" computation: more or less parallel data processing where the need is to solve a specifiable problem. There are biological modelling applications: neural network simulations where the need is to approximate, hence to understand, brain function. And there are embedded systems applications: sensing and control tasks where the need is autonomous dynamic response in changing environmental conditions, often within power and area constraints.

2.3.1 Computing Applications of Neural Hardware

Neural networks have already proven useful for some standard computer applications. In fields such as economic modelling [ASAW06], fraud detection [GR94], and image or character recognition [Kli90], [MGS05] commercial neural network software has existed for years. Generally, however, where such applications have been successful is when they can be transformed into an off-line process running as conventional clocked sequential algorithm [GT02]. In software, neural networks run relatively slowly and may not provide real-time dynamic response, e.g. a software-based neural network fraud detection system is unlikely to be able to provide on-line credit card approval at the point of sale, not with millions of transactions occurring simultaneously. In such an application, requiring real-time response with large data streams, an event-driven model is more appropriate [GSW02].

Conventional hardware is a poor architectural match for the event-driven model, suggesting a dedicated chip [JSR+97], [JH92]. However, this requires a large initial investment for a technology with substantial risks [MLS04]. Compared against the virtual certainty of progress in conventional processing technology during the time needed to design and fabricate the chips, this commitment usually looks unjustifiable [NG03], [ICK96]. There is no reason to expect that a chip designed for a specific neural network will perform well with a random application, certainly not that it will perform well in applications beyond a target application for which it was designed, and so pre-existing neural chips offer no definite benefits to the new user and few opportunities for reuse to the user already having such hardware [Rey03]. If, however, there were a system that allowed easy event-driven applications development for standardised neural hardware, neural computation would look increasingly attractive for large-scale real-time applications. Users could invest in a single system which would then be reusable for future applications and which provided an effective alternative computing resource to traditional mainframes.

2.3.2 Hardware for Biological Simulation

Further progress in understanding biological neural networks virtually mandates dedicated hardware [JL07]. Simulating large-scale biological neural networks using software has become increasingly difficult if not completely impractical, with biologists expecting ever-more detail from ever-larger models [MCL+06]. Widespread use of spiking models has made event-driven processing standard in current software simulators [HC97], [DT03]; however, these run on sequential hardware, resulting in an inefficient layering of emulation: synchronous hardware, simulating event-driven execution, to drive a simulation of a neural network. Unsurprisingly performance is poor [JSR+97]. Meanwhile, there is a mismatch between available neural hardware and the models used in biology. Where the former usually use very simple spiking models such as leaky-integrate-and-fire, (when they implement spiking models at all) [MJH+03], [ICD06], the latter are commonly complex multicompartmental conductance-based models such as Hodgkin-Huxley [HC97], [BPS10]. This points to a gap in mutual understanding: the divergent goals of the hardware designers, who typically are interested in the *nature* of the neural model of computation [WNE97], and the neurobiologists, who tend to be more interested in the *causes* and *effects* of neural computation [YC10].

The typical process of the biological modeller is to build into the model as many properties and components as practicable that have been observed in actual biological studies [MCL+06], [Dur09]. Such an attempt to learn by imitation, however, usually results in a model that yields very poor predictive power. If the model replicates the biological behaviour, all is well and good, but the experiment has not revealed anything the researchers did not already know. Meanwhile if it does not, there is one more mystery unsolved. In either case the researchers gain very little real insight. There can be no understanding of the properties of neural computing, beyond simply cataloguing phenomena, without an understanding of its nature [Ban01].

Hardware designers tend to use the engineers' approach: start with a simple model and progressively add features until it works [FT07], [WNE97]. While this approach readily leads to a series of design abstractions containing testable predictions [Hil97], it is important to differentiate a genuine *abstraction* from a mere *simplification*. An abstraction conveys general computational properties that continue to apply whatever additional detail is added; a simplification may actually change the behaviour quantitatively even at the general level [Maa01]. The problem with current hardware is that it usually provides no method to separate the former from the latter, especially since many chips involve various ad-hoc design decisions taken as an implementation convenience [FSM08], [WNE97]. It is virtually certain that understanding neural computation must needs involve some abstraction of the biology, but what the appropriate abstraction is is not clear [WNE97]

What is needed is a general-purpose neural device that can model neural networks at any level of abstraction, so that while it can model very large-scale networks using simple models, it can at least in principle model detailed conductance-based models and provide meaningful acceleration to biologists using such models [HMH$^+$08b]. Equally imperative is that this device have an integrated tool chain that can plug in with minimal interfacing to biological simulators like NEURON or high-level computational simulators like JNNS, so that modellers are not forced to learn a completely different specification and simulation environment [ASHJ04]. A universal neural network system that integrated with current simulators would permit direct Model A/Model B comparisons [BMD$^+$09], making it feasible for biologists to identify conclusively which computational abstractions are valid and begin to attack the question of the biological model of computation systematically.

2.3.3 Embedded Neural Systems

The embedded space may well prove to be the most fruitful field for neural development [BDCG07], [JM95]. Neural networks as computing devices may work for some important large-scale problems but on the whole have no *a priori* reason to be better than a digital approach [Rey03], and this is furthermore not the central function of neural networks in biology, which exist within the context of a creature interacting in an environment. Embedded computation is much more similar to the real-world tasks biological neural networks perform [JM95], and furthermore there are many applications where conventional microcontroller/DSP-based designs work suboptimally [DLR02]. An embedded neural system would be a device connected to sensors on the one hand and actuators (or controllers) on the other, interacting in real time with its environment.

Event-driven computing is a standard practice in embedded systems [GE97]. Existing tools for event-driven applications [CL05], therefore, should integrate relatively easily with an event-driven neural system. However, tools for embedded systems development have, to date, been scarce and usually proprietary [PA05]. The same is true of neural network systems: the result is that the two very rarely converge.

Embedded systems developers typically work with hard constraints on power, real-time performance [PPM+07], and, perhaps most importantly, cost [HHXW08]. Such developers are usually not hostile to using alternative processing techniques and hardware. In this regard event-driven neural hardware has 2 compelling advantages over software running on conventional hardware: improved speed supporting real-time simulation [ROA+06], and the potential for dramatic power savings. If, however, this comes at the price of greatly increased system cost, it probably will not be a realistic option for the embedded designer [Rey03]. This must be looked at from the perspective of *total* cost: a cheap device which is difficult and time-consuming to develop applications for is just as prohibitive as an expensive device [Kön98]. Effective, simple development tools are therefore a prerequisite for neural hardware to have real impact in the embedded community.

For neural hardware to succeed in embedded applications it will have to combine a hardware architecture flexible enough to implement different networks optimised for different domain-specific applications with a development environment that permits the designer to implement tightly optimised models without excessive ad-hoc design. In spite of the difference in context, these requirements look similar to those of the previous two application categories and indicate that the same basic architecture: an integrated universal neural network hardware/-software system - may be the optimum approach for all neural networks. Having remarked at the beginning of this section that digital systems have converged upon the microprocessor-based platform as the best solution for most cases, it seems that neural network hardware may be in a similar position: that once the “right” platform is found the industry will quickly converge on a standard. Motivation for neural architectures thus moves from *chip* considerations to *system* considerations.

2.4 Chips: Digital, Analogue, Mixed-Signal

To outline the form of an "ideal" neural system, it is necessary to examine these systems in reality: how architecture and application translate into concrete implementation and how far actual systems go towards achieving design goals. This is easiest to do in terms of process technology: digital, analogue, or mixed-signal.

2.4.1 Digital

Digital designs are the easiest to implement, follow the most aggressive process technology roadmaps, enjoy the most extensive tool support, and integrate most easily with external hardware [Rüc01], [MS88]. However, they also use by far the most silicon area, are the most power-hungry, and have the least natural fit with biological prototypes [JG10]. Caught between these opposing poles of equally powerful advantages and disadvantages, digital chips have therefore been exercises in design compromise [KDR07], [ME08]. One reasonably successful solution is the bit-serial pulsed neural network: while early devices such as [MS88] use the technique primarily as a space optimisation, it anticipates the later development of full-spiking models [SAMK02]. Some devices implement a fixed network type to permit aggressive hardware optimisation [YMY$^+$90], however, these are largely devices of the past after commercial ventures encountered decidedly limited success [DAM04]. This should come as little surprise, however, given that it discards perhaps the most decisive advantage of digital technology: arbitrary reprogrammability [JPL98].

Programmable digital hardware, however, has its own problem: it is difficult to justify what is in essence a specialised microprocessor when general-purpose CPU's increase in speed and power each year [Rey03]. Such hardware fights against standardisation, and this virtually ensures low to non-existent tool support [ICK96]. This made the FPGA approach appear promising for a considerable time: standardised tools operating on standardised hardware [PMP$^+$05], most commonly Xilinx Virtex devices [PWKR02b] but also various Altera [BHS98] and Actel parts [KA95], made it appear as though FPGA's and their associated tools could become a form of standard platform with different models becoming different configurations that could be released as downloadable VHDL or Verilog [THGG07]. Unfortunately the scalability problems get worse, not better, with larger devices [MMG$^+$07], and a tendency of the device manufacturers to limit information on the configuration bitstream [KdlTRJ06] have confined FPGA's to prototyping applications.

The other major approach has been to use off-the-shelf parallel processors. An interesting first generation mapped neural networks on early parallel chips like the Transputer [KM94], Datawave [JH92] and WARP [PGDK88] and identified important architectural issues. However, the subsequent disappearance of these chips, in large part because parallel programming turned out to be difficult [MECG06], meant these research directions were not pursued for some time. More recently, there has been a revival of the idea of mapping neural networks to general-purpose parallel processors because of the emergence of Graphics Processing Unit (GPU) chips, a special-purpose architecture designed for massively parallel matrix computations.

GPU's offer the benefit of highly parallel, standard CPU's within a chip multiprocessor architecture designed for high concurrency and internal communications bandwidth [NDKN07]. GPU's have almost as aggressive a process technology roadmap as FPGA's [LLW05]. Furthermore, graphics chip designers like nVidia have released architectural standards to simplify the task of developing applications for these chips. However, it also represents an attempt to use a device in ways not originally intended; experience suggests such attempts run into unexpected limitations and behaviours by trying to co-opt hardware for purposes outside its design goal [Pra08]. More importantly, existing uses limit the scope for neural optimisation or standardisation: the hardware, certainly, is standard, and so are the tools, but standardised *for synchronous coherent applications* [BK06]. The central *idea* of both FPGA and GPU neural implementations seems sound: use a standard, parallel digital device with generic internal components. However, the actual *implementation* in both cases appears to be an attempt to force-fit an unsuitable device [HMH$^+$08b]. To open a realistic path to standard neural hardware, what is necessary is a device designed around neural networks from the outset, containing programmable on-board standardised neural processing elements and scalable FPGA-like reconfigurability.

2.4.2 Analogue

Analogue designs are compact and power-efficient, and can run at reasonable speeds [ICD06], but the primary motivation for using them has always been that it is possible to fabricate analogue devices whose characteristics have real similarities with biological neurons [HB06]. An analogue chip can not merely "simulate" a neural network, it can *be* a neural network, physically implemented in silicon [ZB06]. However, this intuitively elegant direct-implementation capability has also been conceptually limiting, because as a result most analogue chips directly implement a specific model in hardware [HP92], [Cau96], [LS92], [PRAP97] discarding out of hand the possibility of implementing different neural models with the same hardware. "Direct implementation" is in fact a mesmerising but misleading concept, because the "real" model of neural processing is unknown (except in certain narrow, well-studied areas such as the retina), meaning that outside such special cases there are no certainties an analogue device is directly implementing *anything* [FT07], [TL09]. This forces the neurobiologist to ask hard questions about what biological effects the electronic behaviour reproduces, and whether they are adequate to capture all important behaviour [YC10]. Meanwhile, the classical applications developer or embedded system designer will need to see compelling benefit from the hardware as opposed to off-the-shelf digital devices [Rey03]. Direct implementation places the burden of proof squarely on the chip designers, making standardisation unlikely since this requires broad consensus on the utility of the standard.

A hardwired and often proprietary design effectively puts any systematic attempt at model exploration out of reach. Therefore newer designs attempt to achieve at least some level of programmability and general-purpose use [KGH09], [VMVC07] [WD08]. The introduction of programmable analogue memories, chiefly floating-gate based [HTCB89] [DHMM97], [SKIO95] has eliminated what was historically the most vexing barrier: weight programmability, but programmable analogue *circuitry* is still very much in its infancy [LL10] [FGH06], [DBC06] a long way off standardisation. Furthermore, there are very few tools for working with such devices (except at the very lowest level) [BÖD$^+$99] [BD08], nor is much research being done in analogue tool support. Overall, then, pure-analogue designs seem unlikely at least in the short term to become standard neural components, and if they are to have any future, tools for high-level neural specification and modelling on programmable hardware are a vital research topic [MMG$^+$07].

2.4.3 Mixed-Signal

Mixed-signal devices are the obvious solution to the obvious shortcomings of both analogue and digital technologies, and the focus of many recent efforts. Mixed-signal devices can achieve unusual combinations of speed, device density, and interface simplicity [FSM08]. Superficially, the principal problems would appear to be greater design complexity and reduced tool support [BRE94], but in fact the need to provide an on-chip analogue-digital interface completely alters the nature of the design tradeoff. High-precision ADC's and DAC's are area-expensive [MAM05], and in particular tend to slow the design because of limited sampling rates [SHMS04]. This presents the designer with a difficult design tradeoff between signalling rate, area and usable precision. Limiting the number of converters is the most common choice but this means having to serialise the data streams [SFM08], reducing speed and parallelism.

With *spiking* networks it is possible to abstract the actual signalling to a zero-time point process, and this forms the basis of the emerging neural data serialisation standard: Address-Event Representation (AER) [LWM$^+$93]. AER uses packets that encode the source of the spike as an address and is a proven, efficient way to serialise and then multiplex multiple neural signals onto the same series of lines [Boa00] while making the converters themselves trivial [VMVC07]. It is an elegant solution that overcomes most of the interface problems associated with mixed-signal devices with only one important limitation, the restriction to spiking networks [LBJMLBCB06]. From the point of view of potential for standardisation, AER is not only an outstanding candidate but is in fact well established on the way to becoming a defined standard [CWL$^+$07], thus making it overwhelmingly the signalling method of choice for future neural designs. Interestingly, nothing limits AER signalling to mixed-signal devices [CMAJSGLB08], [LBLBJMCB05], and thus it can be the basis of a "template" for neural hardware. The system that emerges is a chip containing blocks of configurable functionality, possibly mixed-signal [TBS$^+$06], embedded in a connectivity network using AER signalling, using a standard configuration and modelling tool chain that injects both data and configuration control information onto the device using AER packets [LMAB06]: the "neuromimetic" architecture [RTB$^+$07].

2.5 The State of the Art: A Summary

Thus far, neural hardware has been an exercise in design tradeoffs: architectural, applicational, implementational. The development "tree" looks like this:

Architectures

Hardwired

Literal implementations using custom silicon to implement classical computational neural networks. While they offer high acceleration, model support is extremely restricted, often to only one network type, and thus such chips have been mostly a pure research exercise.

FPGA

Flexible implementations using the universal configurability of FPGA devices to offer wide choice of models. These solve the problem of finite model choice elegantly, and reconfigurability can also address questions of dynamic network restructuring and resource utilisation. Scalability, however, remains a daunting barrier due to power and wire routing limitations. The circuit-switched design of the FPGA is a major limiting factor.

Neuromorphic

Implementations, usually spiking, that attempt to achieve some level of biological realism. Recently such designs have also begun to acknowledge the need for configurability. Analogue designs, however, complicate this process because of limited and often low-precision tunability. Digital neuromorphic designs, meanwhile, face serious questions of biological plausibility.

Applications

Computational

Neural networks have already proven useful in computational applications like fraud detection and classification where the system must produce a decision based upon incomplete or noisy data. Such applications will surely continue to be important in the future, however, a careful assessment of how to map them to hardware is critical when implementing them on hardware systems.

Biological

There is a growing consensus that for large-scale biological simulation hardware is essential. However, it is vitally important to keep in mind that as long as the actual biological model of computation remains unknown, any claim to "direct implementation" of *anything* is unsubstantiatable. Thus hardware for biological simulation must be as flexible as possible, placing the emphasis on providing a platform for exploring various models of computation rather than examining one pre-decided one.

Embedded

Embedded applications are an ideal fit in many cases for neural hardware. They frequently involve the need for dynamic adaptivity to changing external environments, often under tight power constraints. Neural networks perform well in this area and might be suitable alternatives to traditional digital signal processing. Event-driven computing is a common model for embedded applications, and it also matches well with neural networks. The one missing element that would permit wide-scale embedded neural devices is standardisation: common platforms and interfaces that minimise design time and cost to the developer.

2.5.1 Chips

Analogue

Custom IC's using full analogue componentry to implement neural networks. Usually small-scale because of process restrictions. Difficult to tune, but they do offer the closest implementation to real biology at the lowest power.

Digital

Chips that use standard digital components to simulate neural circuits. Not as space-efficient as analogue designs, but much smaller process technologies offset this, and the chips are vastly more configurable. However, synchronous clocked design increases power dramatically and imposes its own processing assumptions which can be difficult to circumvent.

Mixed-Signal

Chips that attempt some combination of both technologies in order to get a "best-of-both-worlds" design. Exotic process technologies and complex design make such chips challenging, however, they are notable for introducing the AER standard that provides a simple model for "building-block"-style implementation of multichip neural systems and permit the emerging "neuromimetic" hardware generation.

Chapter 3

A Universal Neural Network Chip

3.1 Overview

So long as model exploration remains at least as important for neural networks as hardware acceleration, configurable special-purpose hardware will appear attractive. While it would be possible to examine architectures for neural systems as a purely theoretical exercise, it is far preferable to explore ideas about neural network implementation in the context of a specific, real hardware platform. Real working hardware makes implementation decisions concrete and clarifies the nature of design decisions. Here the discussion uses the SpiNNaker chip as a specific example to introduce the "neuromimetic" architecture. The first part outlines the critical features of the neuromimetic architecture. In essence, these are the defining attributes of neural networks: native parallelism, event-driven update, incoherent memory, and incremental reconfiguration. Each of these attributes has a corresponding specific implementation on SpiNNaker which is central to its function and helps to illustrate the overall architecture.

The second part elaborates the essential theme of abstraction, without which understanding of the neural model of computation will remain purely empirical. Neural networks present three dimensions of abstraction: topological, computational, and temporal. Of these, the need for topological abstraction is perhaps most accepted and understood, but also subjected to the most frequent oversimplifications. Computational abstraction is a more contentious issue: is there a "canonical" neural process? Until a satisfactory theory of neural processing exists, any attempt to impose a neural model *a priori* will be uninformative. Temporal abstraction is a theme that has only emerged recently but may be one of the most important parts of neural computation. Representing both the biological systems' extraordinary dynamic range of time and its ability to represent time-invariant concepts requires systems that can reconfigure time as easily as space or process.

The third part looks at the SpiNNaker hardware architecture in detail. In particular, it describes the components that implement each major feature of the neuromimetic architecture. In turn these introduce behavioural characteristics that are quite different from conventional systems and will require rethinking the development model. Neuromimetic architectures like SpiNNaker introduce a new design paradigm for neural networks: an event-driven model that represents a significant departure from conventional digital processing, with enough flexibility to model many different classes of network but enough neural-specific features to represent a useful abstraction of neural computation.

3.2 The Neuromimetic Architecture

Concrete realisations of the neuromimetic chip concept that is emerging as a leading architectural model for flexible large-scale neural systems are beginning to appear. One such chip is SpiNNaker, a universal spiking neural chip multiprocessor that represents a fourth-generation hardware architecture [FT07]. The essential architectural feature is signal and process abstraction. Where previous architectures make assumptions, often very constraining, upon the type of network being modelled, SpiNNaker leaves the choice of model in the hands of the designer. In that respect it is similar to the microprocessor, a general-purpose digital device that does not predetermine function. In spite of its general-purpose design, however, the neuromimetic architecture retains enough of the general and specific features of neural networks to be hardware-optimisable for neural applications. These characteristics are:

- Native Parallelism: Neural networks process data in massively parallel, simple operations.
- Event-driven Update: There is no separate instruction stream - the input data drives the process.
- Incoherent Memory: Local memories do not have and cannot employ global sequential update dependencies.
- Incremental Reconfiguration: Each input datum may alter not just the state but also the state space.

SpiNNaker's architecture uses these 4 components to provide a generic substrate for neural processing: a kind of "blank slate".

3.2.1 Native Parallelism

The single most essential and characteristic feature of the neuromimetic architecture is native parallelism. It has 2 important, defining properties: *hardware concurrency* - there are multiple processes occurring at the same time in separate but functionally identical units; and *process independence* - the internal state of

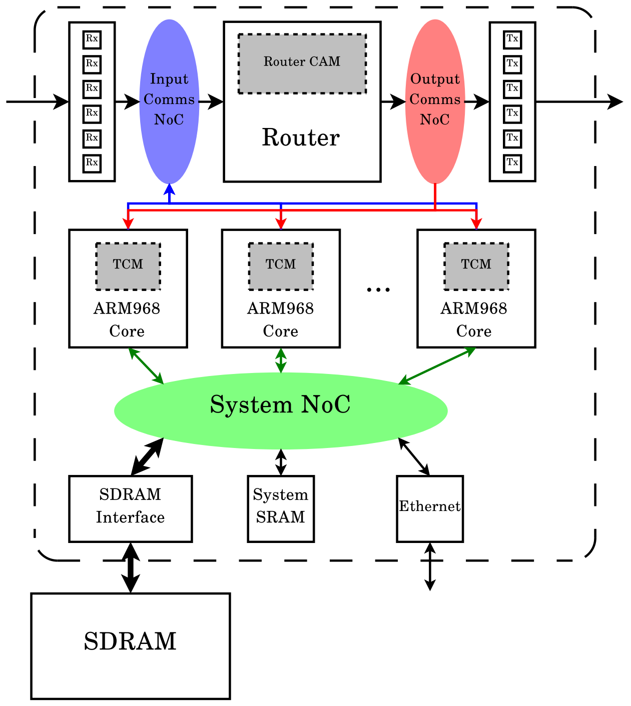


Figure 3.1: SpiNNaker Architecture. The dashed box indicates the extent of the SpiNNaker chip. Dotted grey boxes indicate local memory areas.

one process has no necessary state or timing relationship with another. SpiNNaker (fig. 3.1) achieves hardware concurrency using multiple (2 in the present implementation, 20 in a forthcoming version) processors per chip, each operating entirely independently (in separate time domains) [PFT+07] and having its own private subsystem containing various devices to support neural functionality. This permits a direct physical correspondence between the hardware processors and the individual processing elements (neurons) of the model. The mapping need not be one-to-one, for example for the "reference" Izhikevich model, a given processor nominally models 1024 neurons [JFW08]. However, each neuron resides in an identifiable processor (or an identifiable fixed group of processors) rather than having its processing distributed over the entire system. In this sense, the SpiNNaker neuromimetic architecture is not simply running a simulation on a large system. To achieve process independence, SpiNNaker uses an asynchronous packet-switched network-on-chip (NoC) as the connection fabric [PFT+07]. Programmable distributed routing hardware in the fabric determines the local physical connectivity [WF09]. In much the same way as neurons correspond to processors, axonal connections correspond to network routes, creating a global virtual network that maps the actual connectivity in the neural model. Thus SpiNNaker, uniquely, maintains a direct mapping to the neural model in the configuration of its components while completely virtualising the actual correspondence [KLP+08], so that it implements true native-parallel neural computation without hardwiring the network onto the chip.

3.2.2 Event-Driven Processing

A neuromimetic architecture requires a different model of time and a different control-flow abstraction than the standard sequential synchronous model [RKJ+09]. Individual processors operating independently have no way to know or control when inputs may arrive and thus the input data itself must define the control flow: this is event-driven processing. The defining characteristics of event-driven processing are that an *external, self-contained, instantaneous* signal drives state change in each process, and that each process has a *trigger* that will initiate or alter the process flow, whether it is currently active or not. Thus in contrast to a synchronous sequential system where a clocked counter-driven instruction stream forces an inexorable flow of execution, SpiNNaker's asynchronous event-driven dynamics are a series of context-specific responses to nondeterministic input events [RJG+10].

For the architecture to be useful, there must also be some mechanism to transform the neural process dynamics into an event stream. For *spiking* neural networks the event-driven abstraction is obvious: a spike is an event, and the dynamic equations are the response to each input spike [JFW08]. New input spikes trigger update of the dynamics. Nonspiking networks require different abstractions. Networks with continuously-variable activations can use a neuron that generates an event when its output changes by some fixed amplitude. For models with no time component, the dataflow itself can act as an event: a neuron receives an input event, completes its processing with that input, and sends the output to its target neurons as an event. SpiNNaker has direct hardware support for such event-driven signalling using Address-Event Representation (AER) packets containing the source neuron address (along with, possibly, a data payload). AER packets are *unidirectional.* Destination neurons therefore cannot respond or acknowledge packets received, and implementing reciprocal connections requires 2 separate and distinct AER events. Processors use the familiar interrupt mechanism as the trigger; this also allows the processor to have additional local event sources in the form of support devices that signal via hardware interrupts. Event-driven control ensures that separate processes operate independently, by isolating each process' internal dynamics from the other. This means that the SpiNNaker neuromimetic system is free to follow the time characteristics of its environment and does not superimpose a synthetic clock.

3.2.3 Incoherent Memory

Incoherent memory is a distinctive characteristic of the neuromimetic architecture. Any processing element in SpiNNaker may update any memory variable it has access to without testing for prior access by another processor. Architecturally, this is both a benefit, because the system may as a result dispense with complex memory coherency hardware and protocols, and a necessity, because under the "fire and forget" model memory devices have no way to signal back to a processor the outcome of a request. This affects the structure and organisation of the memory as well as its access mode [RJG+10].

SpiNNaker has two types of memory resource: local and global. Local resources are exclusive to the processor node and therefore inherently have no coherency issues. For global resources, however, applications must ensure that they have no shared read-after-write dependencies, and atomic read-write operations are forbidden. Very much like event-driven design requires any clocks that exist to be local to a given processing element, incoherent memory in effect requires memory updates in SpiNNaker to have purely local dependencies. This is consistent with biology: overwhelming evidence indicates that neural memory appears to reside in synapses, whose weights are only directly visible to their corresponding source-target neural pair [MSK98], [HCR+04]. Global control is confined to low-frequency non-exclusive modulatory pathways whose effect depends only on the local visibility of the modulation [AN00]. In this important aspect, the memory access model of the neuromimetic architecture replicates that of a biological neural network.

3.2.4 Incremental Reconfiguration

The neuromimetic architecture differs from a classical parallel multiprocessor in that the structural configuration of the hardware can change dynamically while the system is running. It is important to distinguish reconfigurability from reprogrammability. The latter requires processing elements to perform explicit access to a general-purpose memory each time the affected processes execute. The former, by contrast, partitions SpiNNaker as a series of independent on-chip resources with hardware that automatically accesses configuration data to change the data and control flows between resources without requiring them to access memory explicitly [KRN$^+$10]. It is possible to use this ability to implement neural elements directly in a one-to-one mapping to resources (as in the reference Izhikevich model), but nothing about it forces any particular relation between hardware and model; indeed, reconfigurability decouples the model from the hardware, allowing experimentation between different hardware to model mappings.

Although not yet implemented in the software model, SpiNNaker hardware supports 2 different types of reconfiguration: dynamic changes by partially or completely reloading the processor memory, and topological changes by reconfiguring the hardware network. Each processor has its own private local memory which contains the neural dynamics, along with dedicated devices that can independently reload this memory without processor control. By requesting a reload of the local neural dynamic memory, an external event can trigger reconfiguration without further control, and since all the devices involved are local, it need not shut down the entire system. Changing the topology is a matter of updating the network fabric's routing tables. The distributed routing structure make is possible to change any local subpath of the network without affecting activity in the rest of the network [NPMA$^+$10]. Boot-time configuration reserves one processor per chip to be able to manage this reconfiguration process locally, thus the remapping can occur independently of the neural processing, just like changing the dynamics [KRN$^+$10]. Incremental reconfiguration in the SpiNNaker neuromimetic architecture achieves 2 purposes: it permits dynamic mapping of the "physical processing" in the hardware to the "virtual processing" of the model, and it allows the user to control the degree to which the hardware directly models the neural network.

3.3 Hardware Abstraction of Time and Space

For large-scale neural models to be intelligible they will need useful abstractions of neural computation. Thus SpiNNaker's neuromimetic architecture comes with the capability to model networks at different levels of structural replication. Indeed, the 4 basic properties of the neuromimetic architecture are abstractions of biological neural computing, which is massively parallel, appears to use event-driven control through asynchronous spikes [MMP$^+$07], has no identifiable memory coherence mechanisms, and supports incremental reconfiguration (through a variety of methods including biochemical modulation [FHK93], cell growth and death, and synaptic pruning and proliferation [BC88]). SpiNNaker is a vehicle for exploring the *computational* significance of neural networks, quite aside from what details it may reveal about the actual processing in biological neural networks or what the choice of model is. Still, to make real headway in deciphering the biological model of computation it is necessary to be able to abstract all aspects of the computing: spatial, temporal, and dynamic.

3.3.1 Topological Abstraction

The need for topological abstraction: that the hardware connectivity need not match physically the model connectivity, is perhaps the most readily understood and already accepted notion of abstraction in the neural hardware community. For the most part, this has been for purely practical reasons: the number of needed connections scales faster than silicon technology can achieve, and with even moderately-sized networks a one-wire-per-connection approach quickly runs out of routing room. FPGA's provide an illuminating case study: the circuit-switched architecture has historically limited FPGA neural implementations to small networks [HMH$^+$08b], and inspired many attempts to extend FPGA limits by various innovative dynamic-mapping techniques [TMM$^+$06]. Just as critically, neural network models do not all have the same type of topology to begin with [UPRS05], and so the wire-per-connection style also ends up limiting model choice. Rather than present the model with a circuit-switched connection infrastructure, SpiNNaker gives all processes access to a packet-switched shared interconnect resource. Nonetheless, at some point this resource also runs out of capacity: available bandwidth sets an upper limit on the number of simultaneously active connections [RJG$^+$10]. Such blockages, however, are temporary, and do not happen at fixed model size. Thus the system can, at least in principle, dynamically adjust the interconnect to obtain the best tradeoff of connection mapping to network utilisation (although this has not been attempted or tested yet). Abstracting topology does more than relax network-size or model-topology limits: it makes it possible to *manage* connectivity intelligently.

3.3.2 Process Abstraction

Differences in neural models are not simply a matter of parameter changes or different terms in the dynamic equations: the very form of the defining equations themselves may be radically different. This means that any hardware that assumes a "class" of equations for neural modelling loses universal power [ŠO03]. For example, typical equations in dynamic spiking models are of the form $\frac{dV}{dt} = E(V) + F(G)$ where V is the membrane voltage, G various cellular conductances, and E and F functions describing their time progression. However, equations in conventional multilayer perceptron models are of the form $T(\Sigma_i w_{ij} S_i)$ where T is a nonlinear transfer function, w_{ij} a weight from source neuron i to destination neuron j, and S_i the input to the synapse. Obviously these are radically different in form, nor are they the only equations involved in such models. Using general-purpose CPU's to implement neurons makes SpiNNaker able (at least in principle) to abstract processes all the way up to arbitrary functions. This involves some associated loss of hardware efficiency, for it is no longer possible as in neuromorphic chips to hard-wire devices to mimic desired behaviour. Given the current (contentious) state of debate over the "correct" neural model [Des97], [Izh04], [VA05] [CRT07] however, such "direct implementation" is probably a false economy. With SpiNNaker, the modeller does not need to answer the question of which neural model is "right" in advance, and furthermore he can continually optimise and refine those models he implements simply by reprogramming. The benefit of process abstraction is as great as it was in the digital case with the microprocessor: the ability to use a general-purpose platform and be able to build a wide range of neural models on top of it.

3.3.3 Temporal Abstraction

Some models have an explicit model of time, others do not; and for those models that do include time there is no necessary relationship between model time, hardware time, and real-world time. If, therefore, the neuromimetic architecture is to conform to the model time in its neural behaviour, it must have internal time representations that are as abstract as those of processes. Synchronous architectures which force explicit timing relationships between processes appear unsuitable for

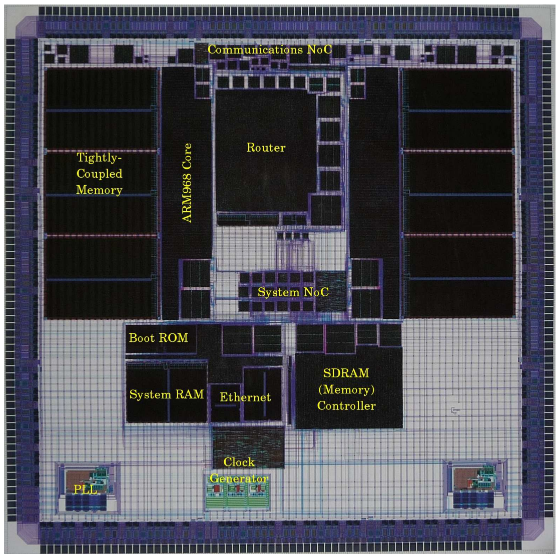


Figure 3.2: SpiNNaker test chip. Functional units are labelled.

such temporal abstraction. Instead, SpiNNaker uses event-driven control flow with asynchronous signalling. An event can happen at *any* time, thus communications do not have an internal time representation that could superimpose itself on the time representation of the model.

It is important to bear in mind, however, that with multiple possible time domains, each communicating asynchronously, the notion of global state on SpiNNaker is meaningless. It is therefore impossible to get an instantaneous "snapshot" of the system, and processors can only use local information to control process flow. In compensation, temporal abstraction brings unprecedented capabilities. Dynamic partial reconfigurability is a natural consequence: since each processing element operates independently, stopping and reconfiguring any one of them does not (and must not) require restarting the neural model. More significantly, temporal abstraction brings the ability to avoid or indeed investigate the effects of clock-based effects on model realism, the ability to run in real time to interact with the world, faster-than-real-time to observe slow learning processes or run computational applications, or slower-than-real-time to examine fine detail in neural behaviour, in different parts of the model. Temporal abstraction does what it says: time is no longer absolute but relative, and while this involves some reformulation of standard models it makes it possible to experiment with time as freely as with function.

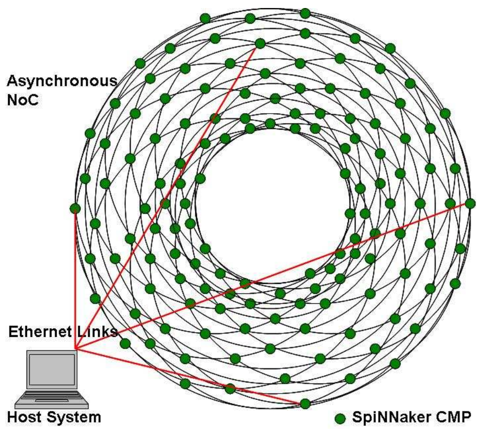


Figure 3.3: Multichip SpiNNaker CMP System.

3.4 SpiNNaker: A Universal Neuromimetic Chip

SpiNNaker (fig. 3.2) uses a combination of off-the-shelf and custom components to implement the neuromimetic architecture. Each chip, a parallel multiprocessor in its own right, can link with others over external links to form a large-scale system containing up to 1.3 million processors over 65,536 chips with high concurrent inter-process communication (6 Gb/s per chip) (fig. 3.3) [KLP+08] Such a system could model a network with $\sim 10^9$ spiking neurons: the scale of a "mouse" or possibly a "cat" brain. Each chip has 2 types of resources, local and global. Local resources are replicated multiple times on-chip, operate in parallel, and are independent from each other. Global resources are unique to a given chip, but each local subsystem can access them independently with non-blocking access. There is one additional important global device residing off-chip: an SDRAM (mostly to store synaptic weights). These global resources, while singular at the chip level, are parallel at the system level: each chip has its own dedicated set of global resources. Thus the whole design of SpiNNaker emphasizes distributed concurrency, the outstanding feature of neural networks.

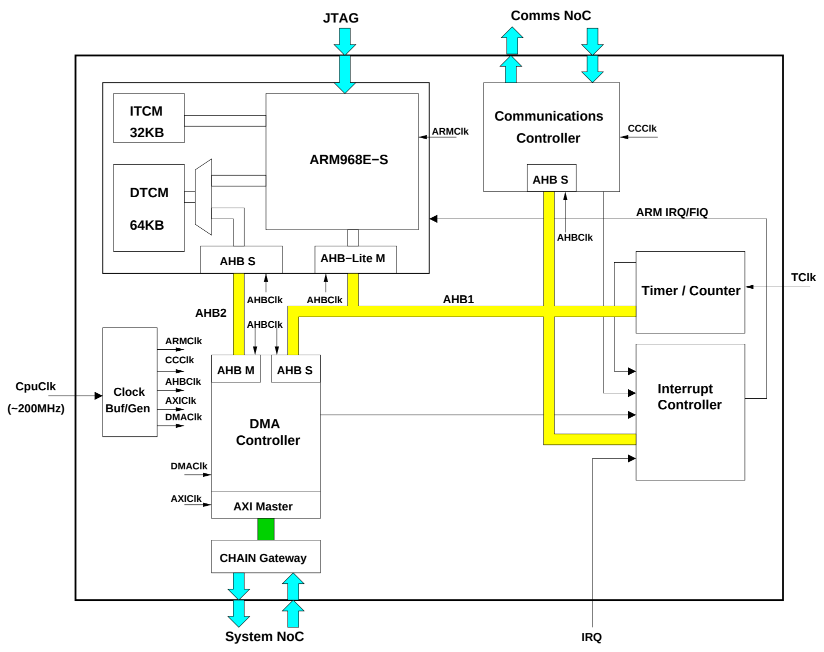


Figure 3.4: Processor node block diagram

3.4.1 The SpiNNaker local processor node

The local processor node (fig. 3.4) is the basic system building block. Each processor node is a complete neural subsystem containing the processor and its associated event-generating devices: a (packet-generating) communications controller, a DMA controller, a hardware timer, and an interrupt controller. While these additional hardware resources optimise the design for spiking models, nothing in the design fixes their use, so, for example, the communications controller can operate simply as a general-purpose packet transceiver. The local node therefore, rather than being a fixed-function, fixed-mapping implementation of a neural network component, appears as a collection of general-purpose event-driven processing resources.

SpiNNaker uses general-purpose low-power ARM968 processors to model the neural dynamics. The processor itself also has a dedicated memory area, the Tightly Coupled Memory (TCM), arranged as a 32K instruction memory (ITCM) and a 64K data memory (DTCM). A single processor implements a group of neurons (from 1 to populations of tens of thousands depending on the complexity of the model and the strictness of the time constraints); running at 200MHz a processor can simulate about 1000 [JFW08] simple yet biologically plausible neurons such as [Izh03]. While the ARM968 is a synchronous core, its clock timing remains completely decoupled from other processors on-chip and from model time. The programmable Timer device facilitates this temporal independence. The obvious way to use the timer is to set it to generate single or repeated absolute real-time events; this is permissible so long as: 1) timesteps (and clocks) remain exclusively local to the processing element they drive; 2) no processing element can require an external event to arrive in a fixed time window. However, another, powerful way to use it that fully preserves the event-driven model is to set the timer on receipt of an input packet event to trigger another event at some programmable delay into the future. This model is in essence a hardware realisation of the event queue common in concurrent modelling software [PKP08].

The embedded communications controller controls AER packet reception and generation for all the neurons being simulated on its associated processor. These AER packets are the only events visible to the neural model itself. Like other devices, the communications controller signals an incoming event using interrupts. The ARM968 supports 2 different interrupt types, "regular" interrupts (IRQ) and preemptive-priority "fast" interrupts (FIQ); assigning the packet event to FIQ logically distinguishes this model-visible event from other, transparent events. The local Interrupt Controller (VIC) centralises event management: all devices direct their interrupts to the VIC which forwards them on to the processor. The VIC provides 2 essential management functions, event (interrupt) prioritisation and interrupt service (ISR) vectoring. Since the VIC is fully programmable, both event priorities and service routines are arbitrarily (re)configurable to suit model needs. In principle this could event include assigning a different event than packet received to FIQ, but in practice this logical division is so useful that management will be more likely to reorder the relative priority between transparent events.

The DMA controller interacts strongly with the TCM. It provides an independent path to local processor memory that transfers data in and out without requiring explicit processor control. The primary function of the DMA is to swap blocks of memory from off-chip storage into local TCM. The most common use of the DMA controller is to transfer synaptic weight data, which typically exceeds the size of the TCM due to the number of synapses involved. The DMA device makes weights appear "virtually local" to the processor by swapping the data associated with a specific input into the TCM when needed, usually after receiving an AER packet [RYKF08]. Synaptic data thus appears to reside in TCM even though in fact only a small slice of it is actually present at any one time. The virtually-local memory model allows memory incoherency: because a given synapse belongs to a definite neuron residing in a single processor, DMA can write back changes to memory without snooping other processors' state, since only the owning processor will access the particular synapse. The DMA controller transforms a synchronous memory access model into an event-driven model by signalling transfer completion with an interrupt, the complete sequence being that an input packet triggers a DMA request process, which completes the transfer independently and then signals the processor using its transfer complete event. The DMA controller is also the only device within the local node with direct visibility of global resources. Thus it is a bridge between the local subsystem and the global subsystem.

3.4.2 SpiNNaker global resources

SpiNNaker global resources implement parts of the neural model that have a system-wide character not easily identified with a specific processing element. These are mainly to do with implementing the connectivity: the Communications Network-on-Chip (Comms NoC), its associated router, and the external SDRAM with associated controller. A few remaining components implement low-level system support features but these do not concern us here; they are invisible to the neural model and functionally independent of it. It is the connectivity information that truly resides in a distributed global resource.

Central to this resource is the Comms NoC [PFT+07]. This asynchronous fabric connects each processor core on a chip to the others, and each chip with its six neighbour chips over links supporting 2 Gb/s internally and 1 Gb/s externally per link. Individual chips' Comms NoC's link to form a global network where a processor is a node. By wrapping links around into a toroidal mesh, it is possible to build a system of any desired scale (up to the addressing limit of the packets) (fig. 3.3). Given the known scalability and routability limitations of circuit-switched architectures [MMG+07], SpiNNaker's network uses a packet-switched approach. Most neural traffic takes the form of "spikes", AER event packets containing the address of the source neuron, and possibly a 32-bit data payload (fig. 3.5). AER allows packets to be source-routed over the network from originating neuron to destinations. Such behaviour matches biology, where

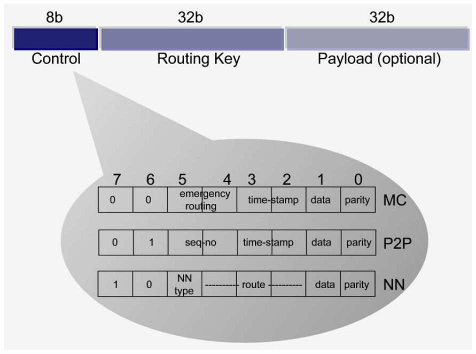

8b
32b
32b
Control
Routing Key
Payload (optional)
7 6 5 4 3 2 1 0
0 | 0 | emergency routing | time-stamp | data | parity MC
0 | 1 | seq-no | time-stamp | data | parity P2P
1 | 0 | NN type | --------- route --------- | data | parity NN

Figure 3.5: SpiNNaker AER spike packet format. Spike packets are usually type MC. Types P2P and NN are typically for system functions.

there is no immediate “acknowledgement” from a destination neuron to source neurons. Because the NoC is a packet-switched system, the physical topology of the hardware is completely independent of the network topology of the model, and it ensures that system bandwidth rather than wire density dominates the scaling limitations. Since the Comms NoC is asynchronous, there is no deterministic relation between packet transmission time at the source neuron and its arrival time at the destination(s). A SpiNNaker system can map (virtually) any neural topology or combination of topologies to the hardware with user-programmable temporal model.

The switching hub of the NoC is a programmable on-chip multicast router that directs packets (spikes) to internal on-chip processing cores and external chip-to-chip links using source-based associative routing [WF09]. This configurable router contains 1024 96-bit associative routing entries to map physical links into neural connections. A default “straight through” routing protocol for unmatched inputs in combination with hierarchical address organisation minimises the number of required entries in the table. The routing table is reprogrammable entry-by-entry on the fly, similar to the “dynamic reconfigurability” of certain modern FPGA’s. By configuring the routing tables (using a process akin to configuring an FPGA) [BLP$^+$09], the user can implement a neural model with arbitrary network connectivity on a SpiNNaker system. With such a programmable router-based network, SpiNNaker achieves the essential feature of incremental reconfigurability while remaining scalable up to large network sizes.

Placing the large amount of memory required for synapse data on-chip would consume excessive chip area. Instead, SpiNNaker uses an off-the-shelf SDRAM device as the physical memory store and implements a linked chain of components on-chip to make synapse data appear "virtually local" by swapping it between global memory and local memory within the interval between events that the data is needed. The critical components in this path are the SDRAM controller: an off-the-shelf ARM PL340, an internal asynchronous Network-on-Chip: the System NoC, connecting master devices (the processors and router) with slave memory resources at 1GB/s bandwidth, and the previously-described local DMA controller. This "synapse channel" uses a memory-swapping method to make data appear continuously local, using DMA to transfer required synaptic weights from global to local memory [RYKF08]. Global SDRAM contains the synaptic data (and possibly other large data structures whose need can be triggered by an event). Since synapses in the SDRAM always connect 2 specific neurons, which themselves individually map to a single processor (not necessarily the same for both neurons), it is possible to segment the SDRAM into discrete regions for each processor, grouped by postsynaptic neuron since incoming spikes carry presynaptic neuron information in the address field. This obviates the need for coherence checking because only one processor node will access a given address range. Analogous to the process virtualisation the processor node achieves for neurons, the synapse channel achieves memory virtualisation by mapping synaptic data into a shared memory space, and therefore not only can SpiNNaker implement multiple heterogeneous synapse models, it can place these synapses anywhere in the system and with arbitrary associativity.

3.4.3 Nondeterministic process dynamics

While this event-driven solution is far more flexible and scalable than either synchronous or circuit-switched systems, it presents significant implementation challenges demanding new methods to transform a neural model into an appropriate, efficient SpiNNaker instantiation.

Specific embodiment of general-purpose processors: Although the processing node is general-purpose, it obviously has a definite hardware implementation. There are therefore better and worse ways to implement a given neural model.

No intrinsic time representation: Events contain no absolute time reference. Particularly with neural models having temporal dynamics, the system must transform an event stream into a definite time sequence.

Arbitrary network mapping: The topology of the communications NoC is independent of the topology of the neural network. Efficient mappings that balance the number of routing table entries required against network utilisation are therefore essential.

No instantaneous global state: Since communications are asynchronous the notion of global state is meaningless. It is therefore impossible to get an instantaneous "snapshot" of the system, and processors can only use local information to control process flow.

One-way communication: The network is source-routed. From the point of view of the source, the transmission is "fire-and-forget": it can expect no response to its packet. From the point of view of the destination, the transmission is "use-it-or-lose-it": either it must process the incoming packet immediately, or drop it.

No processor can be prevented from issuing a packet: Since there is no global information and no return information from destinations, no source could wait indefinitely to transmit. To prevent deadlock, therefore, processors must be able to transmit in finite time.

Limited time to process a packet at destination: Similar considerations at the destination mean that it cannot wait indefinitely to accept incoming packets. There is therefore a finite time to process any incoming packet.

Finite and unbuffered local network capacity: Notwithstanding the previous requirements, the network is a physical interconnect with finite bandwidth, and critically, no buffering. Thus the only management options to local congestion are network rerouting and destination buffering.

No shared-resource admission control: Processors have access to shared resources but since each one is temporally independent, there can be no mechanism to prevent conflicting accesses. Therefore the memory model is incoherent.

These behaviours, decisively different from what is typical in synchronous sequential or parallel systems, require a correspondingly different software model, as much a part of the neuromimetic *system* as the hardware. This model provides important insights about neural computation, and more generally about concurrent computing.

3.5 Summary of the SpiNNaker Neuromimetic Architecture

SpiNNaker integrates the essential elements of the neuromimetic architecture: a hardware model designed to support flexibility in model exploration while implementing as many known features of the neural model of computation explicitly in hardware for maximal performance. The most fundamental of these known features are:

Native Parallelism

Parallel processing is basic to neural computation and thus a neuromimetic architecture must incorporate massive parallelism. SpiNNaker achieves this with a chip multiprocessor design using general-purpose ARM968 processors. These CPU's provide the main facility for process abstraction.

Event-Driven Update

"Real" neural networks use spikes; it is reasonable to abstract a spike to an event and so form the basis for the second feature of neuromimetic computation. SpiNNaker supports spike-based events using the emerging AER packet communications standard, but also incorporates a programmable vectored interrupt controller that permits a wide variety of different forms of event. Because events themselves are essentially "timeless", they allow for easy temporal abstraction.

Incoherent Memory

The notion of memory coherence is irrelevant for biological neural networks, and indeed the need for any coherence mechanism in a parallel system would imply some underlying sequentiality. SpiNNaker therefore has no hardware support for memory synchronisation, merely local and global memory resources that each processor on a chip may access asynchronously. Incoherent memory also eliminates forced timing relationships, simplifying temporal and process abstraction.

Incremental Reconfiguration

Biological neural network structures reconfigure themselves while in operation using several different mechanisms. A neuromimetic system thus incorporates support for dynamic reconfiguration. In SpiNNaker, the principal component implementing this functionality is the programmable multicast router, but in addition it is possible to reload processor instruction and data memory and so change out not just the topology but also internal model parameters. SpiNNaker provides a facility for complete structural abstraction that goes beyond the basic requirement of abstracting the network topology.

Chapter 4

The Library Architecture and Tools

4.1 Roadmap

Functional, effective development tools are essential for hardware to be useful. A potential user needs to be able to access the hardware in ways relevant to its functionality that at the same time offer enough abstraction to provide a clear link to the target applications. This chapter discusses the creation of tools, in particular a neural modelling library, appropriate for the neuromimetic architecture. Reflecting the substantial hardware difference from conventional computing, the library introduces a different, event-driven model. This makes the tool chain as much a part of the neuromimetic *system* as the hardware.

The first part defines the overall software model, highlighting a distinction in neuromimetic systems between configuration and application. Three different components handle different aspects of the running system. “System software” provides essential services as a library of low-level functions; invisible to the user these provide the first level of hardware abstraction. “Modelling software” allows users to configure a model to run on SpiNNaker. It does not run the model but rather sets up the chips to run a model starting from a high-level definition. “User software” actually runs the model and provides an environment, either real or synthetic, for the neural network to interact with.

The second part introduces an event-driven model for tool, and in particular, library, design. At present event-driven tools are scarce, complicating development on event-driven hardware like SpiNNaker where there are three important requirements: abstraction, automation, and verification, which require a reasonably accurate representation of the hardware. The third part describes the specific rôle and design of the library system. To reduce the scope of the libraries to a manageable level, it examines what existing tools, notably SystemC and PyNN, can be used for parts of the process. The library-based system is the critical custom component, using an abstract, event-driven function pipeline to describe a neural model using generic functional elements that the user can configure using parameters to specify the exact model and network structure. Such a system can describe and model almost any neural network, independent of, yet targetted towards, the hardware on which it runs.

4.2 An Event-Driven Neuromimetic Configuration Chain

Configuring a neuromimetic device like SpiNNaker to run a specific model is not simply a matter of writing an application program. Being an event-driven system with hardware-level neural features, SpiNNaker is unlikely to find "standard" tools and software designed for general-purpose sequential synchronous systems suitable, at least not out-of-the-box. Furthermore, the model itself is not an isolated component but is integrated into a complex system that presupposes a configuration process to design, configure, and operate it. Thus, a complete system design approach must consider the model library within the entire tool chain and its interactions with the ultimate purpose of the device [Pan01]. Within the system, there is a need for components to express neural behaviour in terms of SpiNNaker's hardware capabilities, while integrating the event-driven model naturally and transparently, and this is where the library architecture fits in.

There is an important distinction to make between a neural network *configuration* and a neural *application*. In the usual sense, an "application" refers to software that achieves the end-use goal of the system. In this sense, on a neuromimetic system, the application is not the neural model instantiated on the hardware per se, but the interaction of this neural network with the environment (either "real" or simulated) that the modeller presents it with. Configuration, on the other hand, determines the computational architecture of a flexible hardware platform. A SpiNNaker configuration describes a neural network (capable of running various possible applications) as actually instantiated on the hardware. Once on the hardware it is essentially static, at least on the timescale order of the application. While the neural application is completely independent of the hardware platform, the configuration is not - and it is a necessary part of the complete system design since without it the hardware's function is undefined. This study therefore focusses on configuration as an integral part of the system architecture.

The purpose of the configuration system is to provide an efficient hardware abstraction: presenting SpiNNaker to the user as a series of real neural network components while presenting neural network components to SpiNNaker in a way that maximises utilisation and efficiency of actual hardware resources. Such a goal naturally suggests a modular, library-based system with hierarchical levels of abstraction [OCR+06] that uses prewritten, component-like blocks of code to map a neural network.

4.2.1 System Components

At the bottom of the hierarchy lies system software: code that, invisible to the user, performs essential tasks necessary for the hardware to operate. This includes any low-level software components that interact directly with hardware at a signal level (such as device drivers), boot and startup code, and background control and monitoring functions (such as memory management). An event-driven system needs a different software model for these components than a traditional operating system model. The most important difference is that in the event-driven model there is a discontinuous flow of control. This has 2 consequences (particularly on SpiNNaker): first, as well as active modes, there is an overall "idle" mode, in which the processor stops; second, an active mode may have several concurrent processes, none of them necessarily children of another active process. A major part of the functionality of event-driven system components is to provide a global view of system resources to independently executing processes.

The architectural model for the SpiNNaker system layer is that of a scheduler and a series of services. Its core component is the event handler itself, which needs to act as an efficient scheduler of self-contained processes. The scheduler is, in essence, an interrupt handler, and it triggers an appropriate service that then executes. For performance reasons, as in any interrupt service routine, the scheduler should be as lightweight as possible: it need only determine what service to run and then trigger the service. If the processor was in idle mode it can trigger the service immediately, however if the processor was in active mode, it needs to save the context of already-running services.

Services have both a global, shared context, and a local, private context. The model uses the global context to communicate information between services, thus the scheduler, or any other service, may asynchronously update global context, which acts as an instantaneous "message" to other services. Therefore, the scheduler only saves local context, private information representing intermediate states and temporary data within an individual process (fig. 4.1).

The libraries implement individual services. Thus, a given model can load only the services it needs. Services themselves are independent modules that simply terminate when complete rather than returning to a "background" process. They

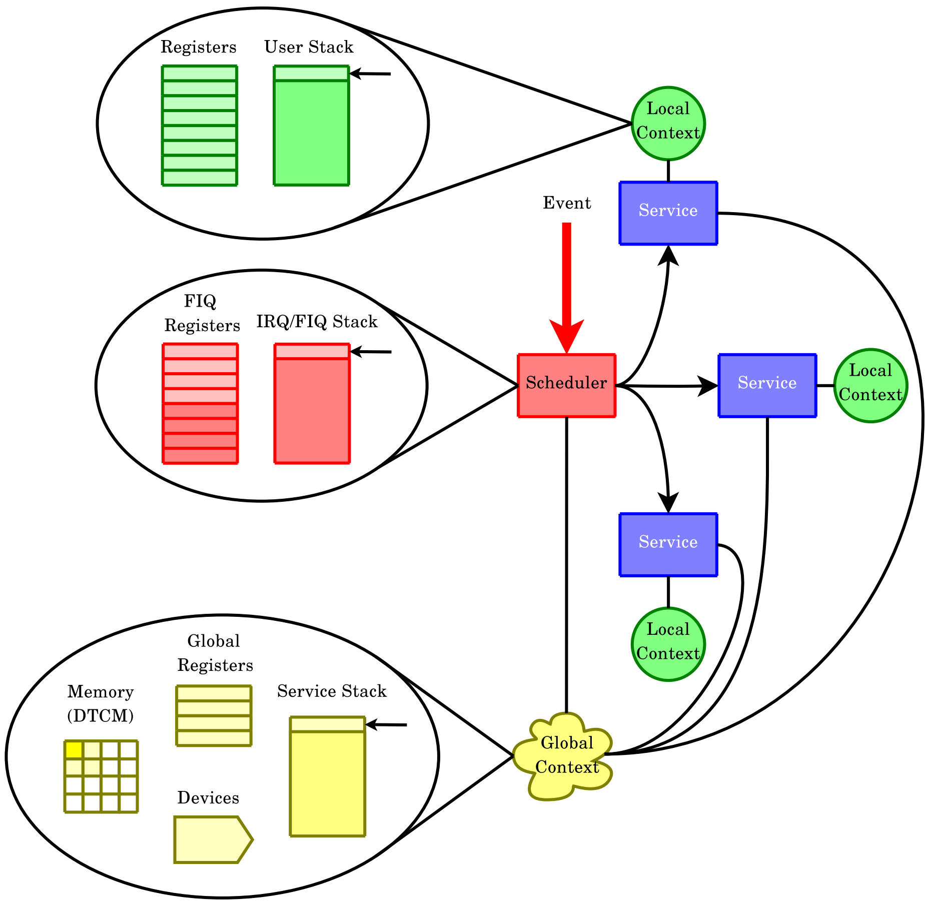


Figure 4.1: SpiNNaker event-driven software model. Events are asynchronous signals that trigger the scheduler: the interrupt service routine. The ISR then in turn triggers various services, which may run concurrently. Services start in ARM Supervisor (SVC) mode but may drop into user (USR) mode after completing critical entry point tasks. Each service has its own local context: the register file and user stack; that the scheduler must preserve. The scheduler does not need to preserve the global context between service calls; it can (asynchronously) update global context, consisting of main memory (DTCM), the service (SVC) stack, device registers, and any registers marked as global. (The number of global registers should be kept to an absolute minimum.) Updates to the global context are the primary method of inter-process communication. The ISR has its own private context. The ARM968 has 2 interrupt modes, FIQ and IRQ; in addition to separate stacks for each, the FIQ mode has its own private registers. Thus no interrupt need preserve scheduler context.

may either provide notification to the system via an event when complete, or simply terminate silently, their action reflected only in the change in global context. If there is no active service, the processor is completely idle: it can "sleep" awaiting a new event. This model reflects the native concurrency both of the hardware and of neural networks, efficiently providing individual processes with a global view of system resources (through the global context), while optimising processor utilisation through an active/idle state model.

4.2.2 Modelling Components

Instantiating a neural model uses modelling components from the library to generate and verify the model on the physical hardware. Starting from a high-level, behavioural network description, the tools need to translate a large model containing possibly millions of neurons into a physical implementation with minimal user direction. SpiNNaker's configurable, event-driven design has 2 differentiating features: *hardware virtualisation* - there is no global fixed map of system resources, and *nondeterministic concurrency* - an event can happen at any time. Together, these mandate *spatiotemporal locality*: processes can only have knowledge of actions in their local execution context. The configuration tools, therefore, divide the neural network into self-contained blocks that are both spatially and temporally independent using an incremental process that requires only local information. The model libraries enable this process by providing atomic units of configuration.

SpiNNaker's component libraries contain predesigned general neural components implementing a particular model type such as a spiking neuron or a delta-rule synapse. These models implement the actual instantiation on SpiNNaker in terms of processor code, memory structures, and configuration commands. At a higher level, other tools: a configurer and mapper, translate the description from neural network software and assemble the SpiNNaker implementation from library blocks. In turn, off-the-shelf verification tools allow the user to simulate the resulting network (a simulation of a simulation!) before running it on the hardware proper. The process of successive abstraction necessary to support a large design leads to a chain of interrelated tools that perform specific subtasks in the translation process from abstract neural network model to concrete SpiNNaker implementation. In this chain the library is the common format for tool interfacing that can express both abstract component descriptions and hardware-identical signal behaviour.

The library's architecture is 3-part: a set of core model-specific functions, a set of memory object definitions, and a set of parameter headers. The functions implement neural and synaptic behaviour and use a generalised abstract neural model defined as a function pipeline. Under this architecture, different neural models have the same general pipeline stages, differing only in the specifics of the implementation of the function representing each stage. Memory objects lie in separate files external from the function definitions, each separate SpiNNaker space (local TCM memory, global System RAM, global SDRAM, and devices) having its own independent definitions. Parameter headers contain constants and initialisation values for variables. The configuration program then assembles a complete SpiNNaker model by inspecting the source model file and extracting neural and synaptic types. It can then build the neuron by retrieving the appropriate functions from the library, together with their associated memory objects. It will plug the functions into the pipeline, configure the sizes of memory objects, and create the parameter headers (possibly with some translation of units) by reading the model's structure and parameter values directly from the source file. The end output is a series of SpiNNaker source files which can be compiled and mapped.

4.2.3 User Components

The externally visible interface is user software: code that, the system having booted and reached an operating state through configuration, the user initiates and controls to run an application. For SpiNNaker the user typically interacts with the system through an external interface to a "host" or "terminal" - a separate computer or processor that provides a suitable user environment. This environment can specify the network model, provide real or simulated environmental input and output, and possibly tune internal parameters or structure interactively. In the ultimate limit, a neural network may need no "user" software at all, being a completely autonomous system connected to suitable sensors and actuators: a fully-functional robot.

Returning to less spectacular implementations, it is probably simplest and most convenient to use an industry-standard PC for the host. While the PC is a conventional synchronous procedural system, various off-the-shelf standards exist that use an interrupt-driven device driver with an internal buffer, solving the event-generation and response problem without the need for extensive custom development. Host interfacing therefore does not form part of the library and is only an external part of the system architecture.

The details of the user host application are not central and may vary from user to user. However, their interfaces to the neural network system should conform to a common interface standard. Recently, PyNN has emerged as a common, cross-platform standard for defining and simulating neural networks. PyNN contains plug-in modules for a wide variety of common neural simulators, and extending it is a matter simply of writing a small interface definition. One present disadvantage with PyNN is that it does not contain definitions for nonspiking network types, but this could be remedied with a PyNN extension where necessary. PyNN provides a ready way of translating between high-level model descriptions and library components, using a plug-in module that associates a given PyNN object with a specific library component. The structure that emerges at the user level is a standard simulation tool, interacting through a common PyNN interface, that enables the user to configure the network and supply it with I/O from the environment.

4.3 Event-Driven Model Requirements

Hardware and software are interdependent and tend to coevolve. Unsurprisingly, therefore, the available tools for design, operation, and applications have become tightly coupled with the expected hardware platforms they coexist with. Using software designed for systems with a different model of computation than the hardware in concern is likely to devolve into an uphill battle of improvisations, yet where possible it is preferable to avoid reinventing the wheel by using off-the-shelf software. This creates problems for event-driven neural hardware such as SpiNNaker. Neural hardware is relatively new and certainly not standardised to any great extent, therefore there are few or no tools designed specifically to interoperate with it [SBS05]. As a result, it has been difficult to develop applications for neural hardware systems or indeed even to get them to run. There has a been a corresponding lack of enthusiasm for neural network hardware in the past.

An event-driven tool chain is thus critical if event-driven systems like SpiNNaker are to be successful. At present few development tools provide clean and integrated support for event-driven processes that makes the flow intuitively clear. A library-based system, however, encapsulates components (which may have arbitrary internal functionality) inside independent modules, permitting an event-driven model that maximises the potential to use existing tools. Under this model, the event-driven dynamics lie in a global, abstract environment into which library components plug in as individual elements. The library components themselves may be written with any suitable development software; only their *interfaces* need conform, or even be visible to, the global environment. If, in addition, this system is not tightly coupled to the SpiNNaker architecture but uses a hierarchy of abstractions that enable modelling on virtually any hardware system, it also solves the problem of event-driven neural networks more generally: a standard for neural modelling in a variety of environments. Such a system maintains a link to traditional synchronous development while demystifying the problem of event-driven design, a truly general-purpose tool chain that is the software embodiment of the universal neural concept of the SpiNNaker chip.

4.3.1 Design Abstraction

Within the library-based architecture, the target application should drive the design flow, not the hardware. Neural network users, in particular, are a very diverse group coming from multiple fields, most with minimal knowledge of or even interest in low-level hardware programming details. The typical user will prefer to input his model in a format similar to how he represents it and simulate it on a system within an environment similar to the interface he uses [BMD+09]. Perforce this mandates *structural* abstraction, so that the tools at the top level can work directly from descriptions written in high-level neural network software [GHMM09]. However, it is equally important for the library system to separate neural events: processes within the model, from hardware events: interrupts in the system, otherwise it would severely restrict model choice, and also complicate debugging because it would be difficult to separate a hardware fault from a model design problem. A tool chain for neural network development on systems like SpiNNaker needs a hierarchy of event abstractions as well as objects. This isolates model-level events (such as spikes) from system-level events (i.e. interrupts) so that the modeller need only contend with behavioural effects while the system designer can successfully program interrupt-level flow without creating unexpected and exotic side effects. The overall effect is a software model with abstract time as well an abstract structure, mirroring SpiNNaker's total abstraction of both at the hardware level.

4.3.2 Design Automation

Manually translating a detailed description of a large-scale neural network containing possibly millions of neurons into a "hardware netlist" would be a virtually hopeless task. An important function, therefore, of the proposed neural hardware configuration tools is design automation: generation of a complete neural hardware instantiation from a high-level behavioural description of the network itself. Ideally, these tools should be able to pass configuration information to simulation tools, and for identical reasons: tracking or executing simulations at a detail level for large neural networks may be too much of a data-processing overhead to be manageable or even useful to a user. The tools therefore must be able to create a full-scale neural network model, translate it to a hardware representation, generate appropriate testbenches, and run simulations with minimal user direction [LBB$^+$04].

SpiNNaker (and indeed, other similar chips) has hardware-specific requirements that influence the design automation tool flow. Library functions must be able to leverage SpiNNaker's neural-network-specific hardware capabilities or the benefit of hardware is lost. This implies the automation process must be able to map high-level neural operations efficiently to device-specific hardware driver routines. Similarly, SpiNNaker's hierarchical memory structure means the tools need to match the type of data with the appropriate memory resource. Most important is packet traffic management. While SpiNNaker has nearly unlimited flexibility to map a network (and hence many possible actual mappings), it has finite communications bandwidth as well as source and destination processing deadlines. The automation process therefore needs to balance packet traffic in both space and time - and this almost certainly involves testing and verification *before* instantiating the model on SpiNNaker, because such effects would extremely difficult to isolate and debug on-chip. The tools need to automate SpiNNaker configuration not simply to accelerate the design entry process but also to find efficient hardware instantiations, because the model to system mapping is (probably) not obvious.

4.3.3 Verification and Evaluation

“A program that has not been tested does not work” [Str97]. With this statement Stroustrup summarises why verification is essential. Verification is a particularly pressing concern and a challenging problem in an event-driven system like SpiNNaker where there is no global clock that can ensure timing alignment. This is a system verification problem, and simply assembling the system and testing it after tends to lead to unpredictable, erratic failures that are hard to debug. Since events can happen at any time, it is very difficult to predict in advance all the possible signal timing scenarios. The SpiNNaker neural hardware, furthermore, does not provide global internal visibility, hence pre-implementation hardware simulation provides visibility that reveals many unexpected cases and exposes subtle dependencies. The library model enhances verification by providing a standard set of functions that can be compared across many models; in turn verification is essential for creating and managing these event-driven libraries so that, long before a network has been instantiated on-chip, its components have been thoroughly tested and debugged.

Verification serves 2 main functions: first, evaluation and confirmation of the neural model itself, second, debugging and benchmarking of the functional implementation on SpiNNaker. This suggests a hierarchy of simulators that can test the system with various levels of process visibility. At a high level, an abstract, hardware-neutral reference model to compare actual (or simulated) chip results against is essential. This should verify both the basic behaviour of the model itself - where there can frequently be some surprises - and the hardware fidelity of the model on SpiNNaker. Such a model does not, however, give enough visibility for library debugging or analysis of system-level behaviour. For this, a second model, that can abstract the hardware enough to avoid obscuring software bugs through lack of symbolic context, yet replicate hardware behaviour enough to verify its functionality with high confidence, is necessary. Native concurrency must be a feature of both models, and their corresponding simulators, to verify event-driven behaviour. The simulation must actually replicate the parallel execution of the SpiNNaker asynchronous hardware, or simulated events would have a hidden synchronous dependency. Using an event-driven paradigm for the libraries makes it easy to translate models across different levels of abstraction in various simulators. The system that emerges is a hierarchical chain of industry-standard tools, using custom, event-driven libraries, with a defined standard interface format containing SpiNNaker-specific semantics that specify particular library templates to use in order to implement (and simulate) particular neural models.

4.4 Design Decisions

The design of the configuration system for SpiNNaker does not propose to build a complete event-driven design and user environment from the ground up: a task of man-years. It is preferable, rather, to exploit the advances made in system tools to date to avoid redesigning complex system components by using off-the-shelf tools where available. Original contributions can then focus on components that must be built from the ground up.

4.4.1 Existing Design and Simulation Software

As it happens, the SpiNNaker tool chain requirements are similar to that of FPGA design, particularly in the need to express and simulate concurrency. FPGA designs typically use a hardware-design-like flow to instantiate components from libraries and connect them according to a high-level behavioural or functional description. The prevailing industry approach uses hardware design languages (HDL's), primarily Verilog and VHDL, to specify a circuit using a softwarelike description, that can be both the input specification for a hardware-generation tool (a "synthesizer") and the functional specification for a verification tool (a "simulator"). HDL simulators rely heavily on an event-driven model of computation to achieve true concurrency. The match with SpiNNaker design requirements is obvious. However, while HDL's can in principle model higher levels of system abstraction, in practice they are specialised for low-level hardware design and do not model hardware/software embedded systems efficiently. HDL's therefore offer a suitable *model* for SpiNNaker development but in their actual *application* are probably limited to chip-level verification.

More recently, system design languages (SDL's), of which SystemC [Pan01] has emerged as a standard, permit design and verification at a higher level of abstraction while retaining the same concurrent properties as their HDL cousins. SystemC contains a built-in simulator that eliminates the need to build one from the ground up. It fills the need for a fast system level simulator [MWBC04], and could also be suitable as an advanced description interface for users wanting finer-grain control over model optimisation that what could be achieved with a library-driven model-level description. However, since in effect, SystemC exposes the underlying simulator architecture to the designer, it remains fairly tightly coupled to the hardware, while making it difficult to compile components for any target other than the simulation. As a result it is probably unsuitable as an abstract model description interface. Nevertheless, SystemC is an excellent match for the complete system design approach - indeed, was developed precisely for this purpose - and is a standard platform for testing model/hardware interactions that does not require building a SpiNNaker simulator from the ground up.

At the topmost level, the obvious choice for a hardware-neutral "reference" model is to create it in any one of several popular (software) neural simulation packages. Typically, the modeller will already be familiar with and using such a package. Neural simulators, unsurprisingly, have the best fit with "actual" neural networks, however, if the choice of simulator for comparison against SpiNNaker is to be *arbitrary*, there are 2 complications. First, such packages come with their own, usually proprietary, simulator and are tightly coupled with it. Secondly, simulators such NEURON [HC97] or GENESIS on the biological side or JNNS or Lens on the computational side make fairly deep assumptions about the type of model, limiting the networks that can be implemented at all, often in ways that complicate the hardware mapping. A new interface, PyNN [DBE+09], attempts to solve this by acting as a simulator front-end, providing the essential network-design and user-interfacing functions while supporting any of a number of different simulators as the back end. This makes it the model-level tool of choice for SpiNNaker. By extending the PyNN development environment with a SpiNNaker interface and a wider variety of models including non-spiking options, it is possible to achieve a seamless verification and automation system. The model libraries themselves operate as the units of translation in an overall system architecture that uses off-the-shelf Verilog, SystemC and PyNN-driven simulation and model description.

4.4.2 The Limits of Reuse

This system has the advantage of maximal reuse of existing tools and a single point of interface at each level of abstraction. In order to provide seamless SpiNNaker functionality, however, there are certain tools that must be built from the ground up . Neither PyNN nor SystemC provide integrated support for design automation. That is, they cannot auto-generate a configuration from an abstract description. SpiNNaker requires executable code, data object definitions, and router table entries that need to be generated from within the tool chain in order to provide a running configuration. To generate these entries, the system needs two additional software components: an object builder (box "Configuration" in fig. 4.3), which constructs the code and data definitions, and a mapper, which assigns individual objects to specific processors, configures the router table entries, and defines the memory map for each processor. Since the intent is to examine the *architecture* of the configuration system rather than emphasize the specific *implementation* the current design considers fairly simple versions of these tools, adequate for $\sim 10 - 100$ chips; larger-scale systems will require more sophisticated implementations that are the subject of continued research. Regardless of implementation, however, the object builder interacts strongly with the library.

In HDL's, the standard way to implement an object builder uses library files to associate pre-built objects with higher-level source files [HM94]; this approach also works well in the SpiNNaker case because it is possible to separate components into code blocks: a C file for the ARM code, a file for each of the memory areas, and a file for the routing table entries. A SpiNNaker model specification is therefore three objects containing neural dynamics, state containers, and connectivity rules, bound to a group of rules specifying how to instantiate these objects. The tool then maps ARM code segments, data segments, and router entries with members in each of the user's specification objects, matching hardware library objects (boxes "Neural Models" and "Synapse Models" in fig. 4.3) against model objects. The result is a SpiNNaker "object file" that the mapper can use to create the top-level neural network that will run on the chip. Importantly, this reduces the rôle of the mapper to pure place-and-route, providing a clean boundary between components of the tool chain. Typically, the object builder translates connectivity rules to router table entries, neural dynamics to blocks of ARM C or assembly code, and state containers to memory objects resident in TCM or SDRAM. A simple script-driven routine completes the configuration description for each processor, using a skeleton main() function with predefined interfaces for neural objects to act as a top-level container for a single running ARM image. The output of the process is a list of available "processor types" and required connections that the mapper uses to perform place and route.

This configuration model has 3 important limitations. First, it assumes the availability of suitable SpiNNaker library files for a given model. These routines, in C or assembly, again have to be built from the ground up. Thus at present the system does not have comprehensive support for every possible model; there are 3 basic model types: Leaky-Integrate-and-Fire and Izhikevich spiking neurons, and sigmoid continuous-activation neurons, together with 3 synaptic types: fast STDP, slow NMDA/GABA-B, and delta-rule. These types, however, are adequate to demonstrate the library *principle*; in future the SpiNNaker group or third-party developers can create other neural types as necessary. Second, it assumes that a single processor implements a single type of neuron and/or synapse. Implementing different neuron types on the same processor is inefficient due to possibly heterogeneous storage requirements and probable context switching. This is not a severe limitation; for example the Izhikevich neuron can model 26 different behaviours simply by a change of parameters, and the sigmoid type has similar richness. Nonetheless modellers need to be aware that some network sizes may map inefficiently in terms of number of neurons per processor. Finally, hardware limitations dictate an absolute number of neurons per processor depending upon the time model. The basic requirement is $N_\nu[T_\nu + (N_w A T_w)] \leq \delta t$, where N_ν is the number of neurons per processor, N_w the number of synapses per neuron, T_ν and T_w the time to update a neuron or synapse respectively, A the mean activity of the network (proportion of active neurons) and δt the modelled time step. Thus if the model has temporal dynamics, there is an interaction between the model complexity (update time), coarseness of temporal resolution (time step), and number of neurons modellable. In extreme cases the hardware may not be able to meet the model requirements, for example expecting real-time update behaviour in multicompartmental Hodgkin-Huxley neurons with 1 ms resolution would not be realistic if 1000 neurons per processor were to be modelled. A hierarchical concurrent description and modelling environment using standard tools in combination with custom libraries realises a universal system to generate an efficient on-chip instantiation, at the cost of possible network constraints in complex or unusual models.

4.4.3 The Neural Function Pipeline

A function library whose models are so different at the system level that they share little or no process commonality makes it difficult to run large-scale networks with heterogeneous internal neural modules, offers no consistent guidance to third-party developers on how to develop new models, has minimal scope for reuse, and is complicated to debug and manage. Therefore a successful function library must incorporate a generic process abstraction that is able to map an arbitrary neural model into a single processing framework.

To meet SpiNNaker's hardware constraints efficiently and accurately, it is useful to define an appropriate abstraction using a set of general design rules that help to define the model implementation. These rules are indicative but not forcing, so that while models generally obey this pattern they can in specific details deviate from it.

Defer event processing with annotated delays

The deferred-event model [RJKF09] is a method to allow event reordering. Under this scheme the system only performs minimal processing at the time of a given event, storing state information in such a way as to be available to a future event, so that processes can wait upon contingent future events. *Future* events thus trigger state update relevant to the *current* event.

Solve differential equations using the Euler method

The Euler method is the simplest general way to solve nonlinear differential equations. In it, the processor updates the equations using a small fixed time step. The method then simply uses the formula $X(t+1) = X(t) + \frac{dx}{dt}(t+1)$. The time step is programmable (nominally 1 ms in the reference models), so modellers can choose finer time steps for better precision or coarser ones for more relaxed timing margins (and potentially more complex models).

Represent most variables using 16-bit values

Various studies indicate that 16-bit precision is adequate for most neural models [DDT+95], [WCIS07]. Since the ARM contains efficient 16-bit operations it makes sense to conserve memory space and use 16-bit variables throughout. Intermediate values, however, may use 32 bits to avoid unnecessary precision loss.

Precompute constant parameters where possible

By an astute choice of representation, it is often possible to transform a set of parameters in a neural equation into a single parameter that can be precomputed. Furthermore, it is often possible to choose this representation in a way that further simplifies the computation remaining. For example, in the expression $x(t) = Ae^{kt}$, it is possible to use the substitution $log_a b = \frac{log_c b}{log_c a}$, choose 2 for c and arrive at $x(t) = A(2^{(log_2 e)kt})$, which makes it straightforward to precompute a new constant $\lambda = klog_2 e$ and determine x with simple shift operations.

Compute non-polynomial functions by lookup table

Lookup tables provide a simple, and in fact the *only* general way of computing an arbitrary function. The ARM takes at most 2N instructions to compute a LUT-based function with N variables. Memory utilisation is a concern; even a 16-bit lookup table requires 64K entries (the entire DTCM) and is therefore impractical. However, an 8-bit lookup table occupies only 256 entries (512 bytes if these are 16-bit values) and can access the value in a single ARM instruction. Where models need greater precision various polynomial interpolations can usually achieve the needed accuracy.

Exploit "free" operations such as shifting

Most ARM instructions can execute conditionally, many arithmetic operations can shift an operand before doing the instruction, and there are built-in multiply-accumulate instructions. Taking advantage of such "free" operations is an obvious optimisation. An extreme example is the SMLAWx instruction. This performs a 32-bit * 16-bit multiply, truncates the low-order 16 bits of the 48-bit result, then accumulates it with a 32-bit value. If, then, the 16-bit value is a number representing a fraction with implied leading 1, by choosing it to represent the reciprocal of an integer it is possible to perform $w = \frac{x}{y} + z$ in a single instruction.

These rules permit the construction of a generalised function pipeline to represent a neural process that is adequate for most models (fig. 4.2).

4.4.4 3-level system

The complete software model (fig. 4.3) defines 3 levels of abstraction: device level, system level, and model level. At the device level, software functions are direct device driver calls written mostly in hand-coded assembly that perform explicit hardware operations without reference to the neural model. The system level abstracts device-level functions to neural network functions, implementing these functions as SpiNNaker-specific operation sequences: templates of neural

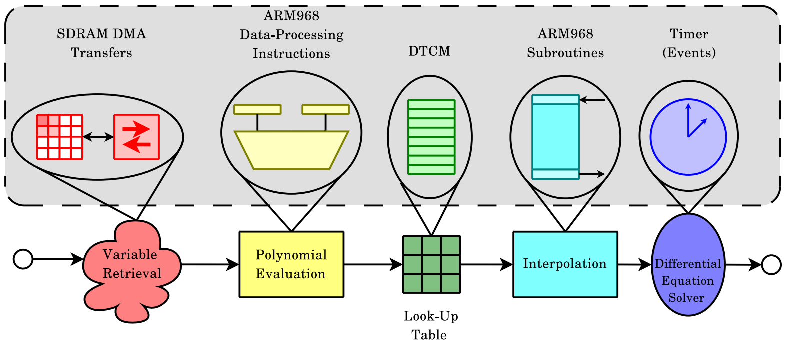


Figure 4.2: A general event-driven function pipeline for neural networks. The grey box is the SpiNNaker realisation. Variable retrieval recovers values stored from deferred-event processes as well as local values. Polynomial evaluation computes simple functions expressible as multiply and accumulate operations. These then can form the input to lookup table evaluation for more complex functions. Polynomial interpolation improves achieved precision where necessary, and then finally the differential equation solver can evaluate the expression (via Euler-method integration). Each of these stages is optional (or evaluates to the identity function).

functionality that invoke a given hardware function. At the model level, there is no reference to SpiNNaker (or any hardware) as such; the modeller describes the network using abstract neural objects that describe broad classes of neural behaviour. In principle a network described at the model level could be instantiated on any hardware or software system, provided the library objects at their corresponding system and device levels existed to "synthesize" the network into the target implementation. This 3-level, SDL-like environment allows modellers to develop at their own level of system and programming familiarity while retaining the native concurrency inherent to neural networks and preserving spatiotemporal relations in the model.

Model level: concurrent generic description

The model level considers a neural network as a process abstraction. The user describes the network as an interaction between 2 types of containers: neural objects ("populations") and synaptic objects ("projections"). Both types of container represent groups of individual components with similar behaviour: for

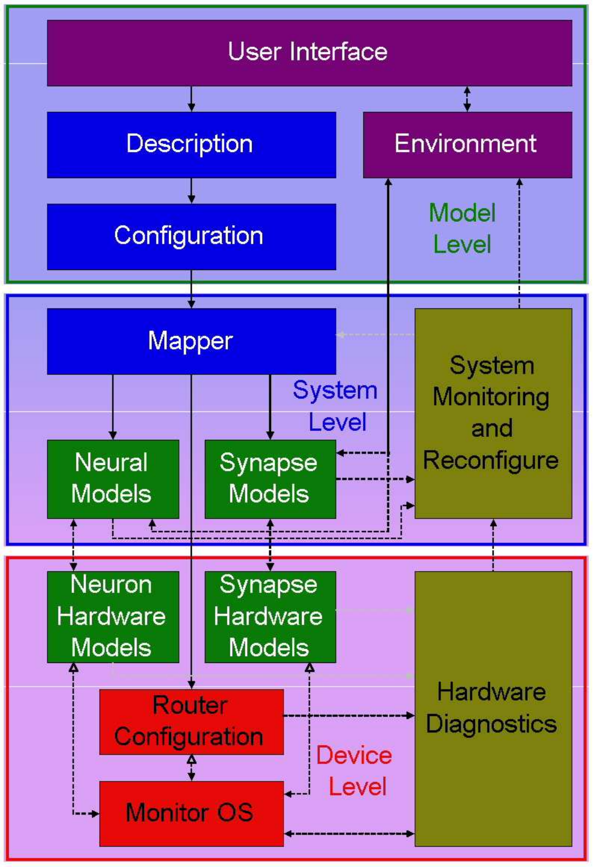


Figure 4.3: SpiNNaker system tool flow. Arrows indicate the direction in which data and files propagate through the system. A solid line represents a file, where dashed lines indicate data objects. Boxes indicate software components, the darker boxes being high-level environment and model definition tools, the lighter ones hardware-interfacing components that possess data about the physical resources on the SpiNNaker chip.

example, a population could represent a group of 100 neurons with identical basic parameters. The terms "neural" and "synaptic" are mostly for convenience; a population need not necessarily describe only neurons. The principal difference between objects is that a projection defines a communication between processes and therefore translates to a SpiNNaker routing resource, while a population translates to a process. These objects take template parameters to describe their functionality: object classes that define the specific function or data container to implement. The most important of these classes are functions (representing a component of neural dynamics), parameter definitions (determining the time model and data representation) and netlists (to represent the connectivity). Thus, for example, the modeller might define a set of differential equations for the dynamic functions, rest and reset voltage parameters, and a connection matrix to generate the netlist, and instantiate the network by creating neural and synaptic objects referencing these classes in their templates. Since at this level the model is a process abstraction, it could run, in principle, on any hardware platform that supports asynchronous communications (as SpiNNaker does) while hiding low-level timing differences between platforms.

System level: template instantiation of library blocks

At the system level, the model developer gains visibility of the neural functions SpiNNaker is able to implement directly. System-level models can be optimised for the actual hardware, and therefore can potentially run faster; however, they run more slowly in software simulation because of the need to invoke hardware-emulation routines. A given system-level library object is a generalised neural object similar to a model-level object, whose specific functionality comes from the template. At the system level, however, a template is a hardware "macro" - for example a function "GetWeights()" that requests a DMA transfer, performs the requisite memory access, and retrieves a series of weights, signalling via an interrupt when complete. Time at the system level is still that of the neural model, reflecting the fact that the only visible hardware event is the input "spike" event. System level descriptions are the source input for the SpiNNaker "synthesis" process: a bridge between the model level and the device level that uses library templates as the link to provide a SpiNNaker hardware abstraction layer.

Device Level: a library of optimised assembly routines

The device level provides direct interfacing to SpiNNaker hardware as a set of event-driven component device drivers. In the standard SpiNNaker application model, events are interrupts to the neural process, triggering an efficient ISR to call the developer-implemented system-level neural dynamic function associated with each event. ISR routines therefore correspond directly to device-level template parameters. The device level exposes internal process events as well as the input spike event, and therefore the programmer specifies a time model by explicit configuration of the hardware devices. Time at device level is the "electronic time" of the system, as opposed to the "real time" of the model. The device driver library forms a common component of the entire library system and includes the functions needed by a neural application where it has to interact with the hardware to model its dynamics. Providing a ready-made library of hardware device drivers gives users access to carefully optimised SpiNNaker neural modelling routines while also presenting a template for low-level applications development should the user need to create his own optimised hardware drivers for high-performance modelling.

For SpiNNaker, the complete process of instantiating a neural network on the chip would go as follows (fig. 4.3): A modeller would specify the model in PyNN as a dynamic-state-connectivity abstraction (in box "User Interface"). This abstraction then forms the input to a "synthesis" process (boxes "Description", "Configuration", and "Mapper") that translates it into a hardware-level configuration file. A configuration utility (boxes "Router Configuration" and "Monitor OS") then loads the file to the device and starts the system. User-interface software (box "Environment") interacts with the actual simulation executing on SpiNNaker to provide the neural network with a simulated or real environment in which it operates and from which the user can observe actual response. The pivotal role of *libraries* in this software model is clear: it is the libraries that provide the actual units of translation or "object semantics" between the high-level tools modellers are used to and the low-level SpiNNaker configuration-time and run-time environment.

4.5 Summary of the SpiNNaker Development Architecture

SpiNNaker is a sufficiently large departure from conventional clock-driven systems as to require a different development model based on event-driven software. Another large-scale distinction is the separation between configuration, which specifies what neural model to instantiate on the chip, and application, which constitutes the task the neural network is to perform and the objective in running the model. Three successive levels of hardware abstraction provide visibility appropriate to different users with different objectives, and support different components of system functionality.

System Components: Device Level

SpiNNaker Device Level components are a set of device-driver libraries that provide critical operating-system-like support: basic functions needed to run the hardware. Low-level simulation is possible down to cycle-accurate fidelity using Verilog if necessary.

Modelling Components: System Level

System Level components are once again libraries, but these library functions implement entire neural models rather than low-level hardware routines. A general-purpose function pipeline defines a generic structure for the library routines, which then can take template parameters to define a specific neural model. SystemC simulation provides transaction-level model verification with reasonably fast performance.

User Software: Model Level

Model Level presents the user with familiar interfaces using, where possible, existing neural simulation tools. The PyNN standard provides a convenient interface, providing high-level abstract model definitions (that link to System Level libraries) which can be run on any of several supported simulators. A modeller can thus describe the neural model and compare results across several simulation platforms including SpiNNaker hardware, using PyNN as a single, Model Level interface.

Chapter 5

Phase I: Definition - Test Networks

5.1 Outline

What neural models can SpiNNaker run? In principle, *any* model, but in practice, some models are easier to implement, and will perform better, than others. This chapter therefore focusses on describing actual implementation of neural models and examining what design choices lead to efficient SpiNNaker implementations.

It is easiest to demonstrate the SpiNNaker library-based development environment, and illustrate its efficiencies as well as design tradeoffs, by introducing 3 existing neural network models for SpiNNaker. In addition, different individuals and groups have contributed to creating these models, providing insight into the path for the potential future integration of third-party modules. Using the function pipeline abstraction, descriptions of the neural models will first describe the implementation of the neurons and synapses themselves with reference to pipeline stages, then describe the operation of the network as a whole.

The three models are the Izhikevich neuron with STDP synapses, the multilayer perceptron model with delta-rule synapses, and the leaky-integrate-and-fire model with NMDA synapses. While not necessarily comprehensive, these are sufficiently different to represent a reasonable cross-section of potential neural implementations while illustrating various design considerations. Most importantly, these models are intended as *reference* implementations: neural function templates that future developers can extend or modify to create a larger model library.

5.2 The Izhikevich/STDP Model

The first model, and perhaps the most representative in terms of the design goals of SpiNNaker, is a simple reinforcement-learning controller using Izhikevich-style neurons and STDP synapses. This important model is the "reference" network used to guide SpiNNaker hardware design decisions and illustrates key general methods for implementing real-time spiking neural networks in hardware. While biologically realistic and potentially scalable up to large simulations of major brain regions, the purpose of this model has been to establish basic SpiNNaker functionality. The priorities are therefore architectural simplicity and the creation of "infrastructure-level" functions other neural models can readily reuse without significant modification.

5.2.1 Neuron

The network uses the Izhikevich model [Izh03] to implement neurons. While not the very simplest spiking neural model, the Izhikevich model has become the reference neuron for SpiNNaker because it is adequately simple, instruction-efficient, and exhibits most of the dynamic properties of biological neurons. Xin Jin describes most of the low-level implementation of the model itself in [JFW08]; this section therefore discusses it only briefly to indicate key properties of its design. Although not originally designed under the function-pipeline model it readily maps to it: a few simple modifications have been sufficient to integrate it into the library system. It is thus possible to walk through the pipeline step-by-step for this basic neuron type.

Variable Retrieval

ARM968 processor cores contain no floating-point support, and since studies [JFW08], [WCIS07], [DDT+95] indicate that 16-bit precision is adequate for most purposes, the model represents all variables and parameters with 16-bit fixed-point variables. The Timer event triggers variable retrieval for the neuron. This retrieves the neuron parameter block containing dynamic parameters, and the current activation. The processor extracts the accumulated activation for the current time from the data structure of one neuron and passes it along with the (static) parameters to the two Izhikevich equations.

Polynomial Evaluation

Polynomial evaluation simply consists of evaluating the equations for the current time. Both of these equations are simple polynomials:

$$\frac{dv}{dt} = v^2 + I - u; \frac{du}{dt} = a(bv - u)$$

where a and b are constant parameters, v is the voltage, I the input current (usually from synapses) and u a lumped “recovery” variable representing the net effect of turn-off gating conductances.

Look-up Table

While the Izhikevich equations themselves are pure polynomials, one part of the evaluation is in fact a (trivial) look-up operation: determination of whether the neuron spikes. The dynamics replicate a spike-like upstroke, but to generate the spike *event* the neuron uses a threshold value and then looks up as a parameter (a lookup table with one entry) the after-spike reset value.

Interpolation

Since the only look-up functions are single-entry, the processor need perform no interpolation. The results of the evaluation pass instead directly to the differential equation solver.

Differential Equation Solution

Using the Euler method provides a simple bolt-on solution which is general for all neural models that take the form of differential equations. For the Izhikevich model, it simply updates v and u at each time step. Implemented in ARM assembly to optimise performance, the model requires 21 instructions per neuron if the neuron does not fire, 33 if the neuron fires, and 50 if the neuron fires and synapses have STDP dynamics. The processor maintains real-time accuracy as long as it can update its neurons before the next time step occurs - nominally 1 ms intervals in the reference model.

5.2.2 Synapse

The synapse heavily uses the deferred-event model because the time a packet arrives does not correspond to the real-time arrival of the spike in the model. Synapses can be either fixed-weight or plastic (variable-weight). How much of a full function pipeline needs to be implemented depends on this characteristic. Currently there are only a few models for plastic synapses in spiking neural networks, of which the overwhelming majority are of the spike-timing dependent plasticity (STDP) family. SpiNNaker STDP synapses use the exponential update model [GKvHW96]. STDP presents a significant challenge in event-driven systems, because it uses correlations between timing of input and output spikes, and thus if an output spike occurs after an input spike, hence after its associated event, the event information will be lost. Critically, in SpiNNaker, the processor

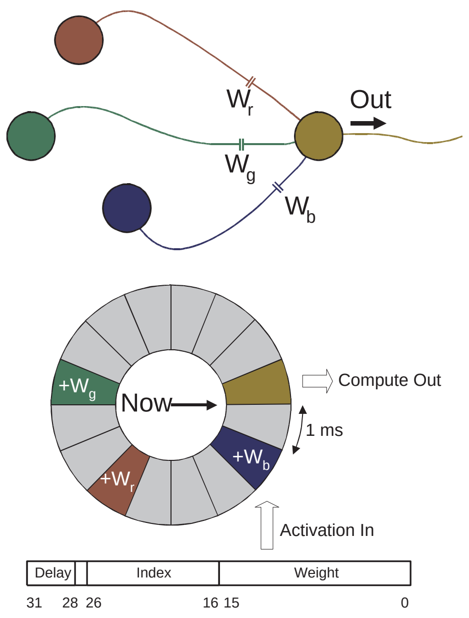


Figure 5.1: SpiNNaker Neuron Binned Input Array. Inputs arrive into the bin corresponding to their respective delay. Output is computed from the bin pointed to by "Now".

uses event information to retrieve synapse data under the virtual synapse model [RYKF08]; by the time an associated output event happens the synaptic data will have been flushed from TCM. Using the deferred-event model, however, it is possible to perform the required synaptic updates without a double data load from memory [RJKF09]. In either case, however, the event that triggers the pipeline is the packet-received event.

Variable Retrieval

Synapse data resides off-chip in SDRAM. Therefore the first action the processor takes when a packet arrives is to request the relevant synaptic data. This triggers the first phase of deferral. The processor initiates a DMA operation which retrieves a source neuron-indexed synaptic row (Details of the DMA process are in [RYKF08] and the data format in [JFW08]). A second stage of deferral happens when the data arrives, because synapses have nonzero delay. To implement the delay, a 16-element array (fig. 5.1) in each neuron represents input stimuli. Each element represents a time bin having model axonal delay from 1 ms to 16 ms. Having retrieved the synaptic values, the process now needs to get the bin corresponding to the axonal delay to a single synapse for each destination neuron in turn. If the synapse has STDP, it must finally retrieve another value from that neuron: a time stamp indicating when the neuron last fired. Each source neuron has an associated "row" in SDRAM containing a time stamp and target-indexed weight entries only for those target neurons on the local processor with nonzero weights (fig. 5.2), permitting a single table lookup per input event to retrieve the synapse data. Time stamps use a 64 ms two-phase time window comparable to the $\sim$ 50 ms experimentally observed STDP time window [MT96], permitting 4 ms spike-time resolution within a 32K $\times$ 64ms coarse time period, enough to capture about 35 minutes without time rollover. At this point the processor has the data it needs to proceed through the pipeline.

Polynomial Evaluation

Evaluation of spike transmission assumes the dynamics are instantaneous, a reasonable approximation for fast AMPA/GABA-A synapses. Given a spike with a delay of δ_m ms arriving at t_n ms, the processor accumulates the weight to the value in the $t_n+\delta_m$ ms time bin. This defers actual spike propagation until that future time.

Look-up Table

Synaptic learning is a look-up table process. For each connected neuron, the processor compares its local coarse time stamp (postsynaptic spiking times) against the retrieved time stamp (presynaptic activation times). If the coarse time stamp matches, the processor performs the weight update rule the model specifies, using the difference in the fine time stamp as the index into a look-up table. (By coding the weight update as the power-of-2 bit-position in the fine time stamp, the processor can compute the update with a series of simple shift-and-add operations). It then updates the time stamp of the presynaptic neuron.

Interpolation

Synaptic transmission requires no interpolation. If precise fidelity to the exponential STDP were desirable, the model could use interpolation to refine the weight modification. However, there is considerable evidence [BP98] that STDP weight updates actually follow a statistical distribution where the theoretical update curve represents the envelope of the distribution. Hence the model does not perform interpolation since it would involve additional computation for questionable increase in biological plausibility.

Differential Equation Solution

Once again, since the process defers synaptic transmission to the *neural* update time, there is no need for a separate solver for the synapse. However, under STDP, if the neuron fires, the processor must update its local time stamp. This algorithm has several useful properties:

1. It gives a long time for processing, generally the real time between input events, which for biologically plausible neural networks is ∼100 ms average-case, 10 ms worst-case.
2. It preserves presynaptic/postsynaptic timing relationships.
3. It provides a 1-instruction test (the coarse time stamp compare) for nonupdated synapses, reducing processing overhead with inactive connections.
4. It works with multiple spike bursts in a short duration, by using the fine time stamp to reduce the number of expensive off-chip memory accesses.

Provided the update computation takes less time than the time between inputs on the same synapse, the model will retain accurate synaptic update. Deferred processing makes it possible to "hide" the update algorithm and memory load/store operations in the time between input activations.

5.2.3 Networks

System context

The test networks for the spiking models form part of an integrated plan to create a large-scale model simulating a significant thalamocortical system. For this model, Izhikevich neurons will be particularly important for control. One of the intended core modules of the large-scale model is a reinforcement-learning control subsystem. Many of the models thus far implemented are subcircuits of this system. It is useful, therefore, to have some context on the ultimate purpose of this planned model. In its simplest form, a reinforcement-learning

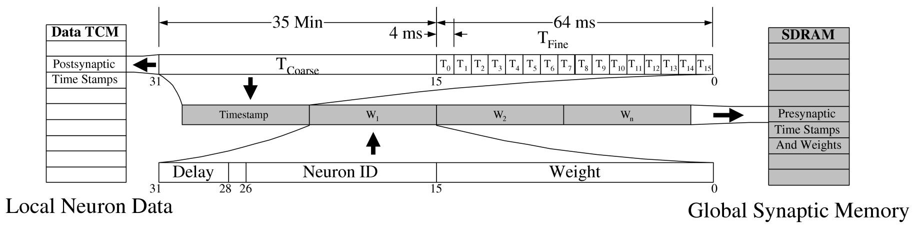


Figure 5.2: Synapse Data Format. Each of the data words is 32 bits wide. Synapse rows in global memory have multiple weights and a time stamp word. Weights contain a 16-bit weight field and a 16-bit parameter field. Experiments used 4 bits of the parameter field to represent synaptic delays and 11 to index the target neuron. Each row in SDRAM has one time stamp (time of last presynaptic event), and each neuron in local memory a second (time of last postsynaptic spike). The time stamp has 2 fields, a fine and coarse time stamp. The fine time stamp is bit-mapped into 16 equal time segments with resolution $\frac{I_w}{16}$, where I_w is the total length of the time window, containing a 1 if there was an input activation at that real-time point within the interval. The coarse time stamp is an integer with a resolution of $\frac{I_w}{\phi}$, where ϕ is a (small) power-of-2 constant representing a time phase within the window I_w. The model updates this time stamp if a transmission event t_{new} occurs at $t_{new} - t_{last} \geq \frac{I_w}{\phi}$, where t_{last} is the current value of the coarse time step.

system contains an "actor" network that processes I/O and takes control action, and a "critic" network that monitors actor responses and forms a prediction of future action likely to lead to the attainment of internal goals, or the avoidance of penalty conditions. With a small modification, the transformation of the goals input into its own network: the "evaluator" having reciprocal plastic connections to the critic (henceforth called the monitor to distinguish it from the basic actor-critic model), this network can build its own goals and supply its own training with only a minimum of hard-wired drives.

The evaluator receives direct input from feature extractors in the Monitor's input pathways. The current system assumes an input with one hardwired, "unconditioned" feature and other, "conditioned" features, not hardwired. The network will have three layers: the hardwired layer, an association layer, and an output reinforcement layer. The hardwired layer will contain three neurons directly connected to the respective inputs with a weight sufficiently large to ensure a strong response to regular spiking input. The purpose of these neurons is to drive targets in the association layer into reliable firing. The association layer itself receives input from a subset of the conditioned features with synapses having random (but small) initial weights and STDP plasticity, and one of the unconditioned neurons with a fixed-weight synapse, its output neurons firing if the frequency and phase of the conditioned and unconditioned features are strongly correlated. In turn, it will drive the output reinforcement layer, which sends bursts of spikes with the characteristic frequency and phase of an active associator output to synchronise the Monitor network. By connecting each group to an associated set of inhibitory neurons that output to all neurons of the group (a WTA: "Winner-Take-All" network), it is possible to ensure that only the most-active output combination within the group responds. This network could therefore function either as a stand-alone delay associator or in its original purpose as the source of external reinforcement to a critic in an actor-critic reinforcement learning network.

Software Simulation

Building this network involves an integrated develop-test methodology; it is now time to turn to the actual test networks implemented in this process. An important part of building and verifying the models has been software simulation using SystemC. A full SystemC simulation of the hardware runs quite slowly ($\sim 15min/ms$); thus it is useful to build simple networks for software simulation, where the aim is not (necessarily) behaviourally relevant network dynamics but rather functional verification of the hardware implementation of the underlying neural and synaptic models.

The first series of tests, to verify neural dynamics and processing speed, use a random network of 60 Izhikevich excitatory neurons. Each neuron connects randomly to 15 other neurons with fixed weights (i.e. synaptic plasticity was off). Weights are random in the range 0-20 mA/mF and delays are random in the range of 1-16 ms. Each neuron receives an initial current injection of 30 mA/mF in the first ms. The simulation ran for 244 ms.

To verify STDP, the software simulations use a second random network of 60 Izhikevich neurons. 48 neurons are excitatory "Regular Spiking" types;12 are inhibitory "Fast Spiking" types. Each neuron in both the excitatory and inhibitory populations connects to 40 neurons selected randomly (this includes the possibility for self-recurrent connections), with random delays in the range 1-16 ms. Connection weights have an initial value of 8 mA/mF for excitatory and a fixed value -4 mA/mF for inhibitory synapses (which are not plastic). Excitatory synapses can reach minimum and maximum values of 0 and 20 mA/mF respectively. The STDP window is set to ±32 ms and the minimum/maximum values of the STDP update are ±0.1 mA/mF. 6 randomly-selected excitatory and 1 inhibitory neuron receive a constant current injection of 20 mA/mF. Simulations stop at 10 s.

Hardware Networks

To test the ability of the SpiNNaker system to run a large simulation in real time, an initial hardware test generated a randomly connected network containing 500 Izhikevich neurons. The network is divided into excitatory and inhibitory populations in a 4:1 ratio, i.e. 400 excitatory "Regular Spiking" (RS) neurons and 100 "Fast Spiking" (FS) inhibitory neurons. Each presynaptic neuron randomly connects to 30 postsynaptic neurons with random delays in the range of 1-16 ms. The network uses fixed weights of 10 mV for excitatory connections and -5 mV for inhibitory connections. 12 excitatory, and 3 inhibitory neurons selected at random receive a constant input of 20 mA/mV. The model runs for 1 second real time.

A core subcomponent of the reinforcement system is the Associator, a simple network that associates a hardwired "unconditioned" reward stimulus with a plastic series of "conditioned" stimuli. Conceptually, the model encodes different conditioned stimuli as a delay value and associates the unconditioned stimulus by matching delays. This network is an ideal candidate for hardware testing because it is very simple and can use synthetic inputs, at least initially. In simple test form, the model is a 3 Layer feedforward network containing 21 Izhikevich neurons. Layer 1 has one hardwired (input) neuron (corresponding to the unconditioned reward). Layer 2 and Layer 3 consist of two populations of 10 neurons each. Layer 1 connects to all neurons in Layer 2 with random delays in the range 1-10 ms and uniform weights of 20 mA/mF. Layer 2 neurons connect to Layer 3 with one-to-one connections having random initial weights in the range 1-10 mA/mF and fixed delay of 1 ms. To simulate a reinforcement signal, Layer 1 receives a constant stimulus of 20 mA/mF starting at 100 ms and ending at 400 ms. The simulation runs for 500 ms before stopping. These networks are designed to show the essential functionality of the library: the ability to instantiate different models on the same chip, and move towards useful subcomponents for a large-scale model. Spiking networks, however, are only one class of model. A general library should also be able to implement nonspiking networks. The discussion therefore now turns to the most common nonspiking type: the multilayer perceptron.

5.3 The Multilayer Perceptron Model

The multilayer perceptron (MLP) is a good demonstration of the universal character of SpiNNaker: in complete contrast to the spiking model it has no intrinsic notion of time, involves graded-response activations (no spiking), and uses learning rules that reverse the direction of synapses: "backpropagation". The MLP is also an important demonstration because it is by far the most widely used model for computational neural networks, thus any chip that cannot implement an MLP efficiently has very dubious claims to being universal. Finally, the MLP is an important driver for tool development, ensuring that in like manner the development system is not limited to spiking neural networks and their derivatives. The MLP requires reconsideration of the entire implementation model.

SpiNNaker's architecture creates two separate implementation considerations: *mapping*: the assignment of processes to SpiNNaker resources, and *dynamics*, the process flow on SpiNNaker. The process flow, however, depends explicitly upon the mapping, which furthermore determines to some extent the identity of what constitutes a "neuron" and a "synapse". Thus it is necessary to consider the mapping problem first.

5.3.1 Neuron

An MLP neuron [1] performs a simple sum-and-threshold function, the threshold function being some continuously-differentiable nonlinear function (usually the sigmoid function $\frac{1}{1+e^{-kS_j}}$). Taken at face value, the obvious implementation would map each unit to a specific processor. The backpropagation algorithm, however, requires bidirectional connections, while SpiNNaker's network is source-routed: the router requires 2 entries to contain both forward and backward mappings C_{ij} and C_{ji} between neurons i and j. This rapidly consumes router entries and increases overall traffic. Worse, the synaptic weights must be stored either at the processor containing neuron i or neuron j. Thus either both neurons would have to reside on the same processor, severely limiting scalability, or signals would

[1] Technically, a "unit" instead of a neuron: in the MLP the unit is a processing element not necessarily associated with a single (biological) neuron.

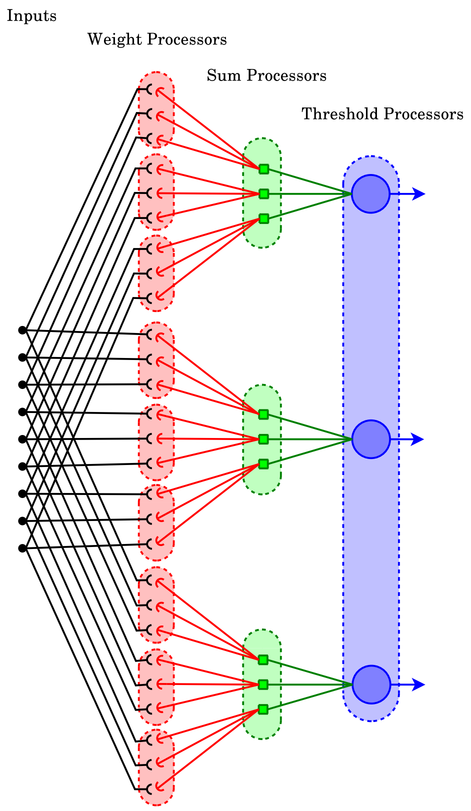


Figure 5.3: SpiNNaker MLP mapping. Each dotted oval is one processor. At each stage the unit sums the contributions from each previous stage. Note that one processor may implement the input path for more than one final output neuron (shown for the threshold stage here but not for other stages)

have to carry the weight information with them. The one-to-one mapping approach would be an extremely inefficient method, making it better to split the processing into weight, sum and threshold components(fig. 5.3).

The model creates a hierarchical compute-and-forward neuron: processors close to their associated synapses compute initial partial sums, then forward these on to higher-level aggregators. At the top level, the threshold units receive the aggregated total and compute the transfer function.

Sum Units

Sum units are trivial: they simply accumulate input packets and output the sum after receiving all inputs contributing to that sum. If the sum components, where possible, lie on the same die as the synaptic processing elements from which they aggregate, the overall mapping further reduces long-distance routing and traffic to a minimum by reducing the $\sim N^2$ synaptic outputs to $\frac{N^2}{A}$, where A is the number of elements a single sum unit accumulates.

Threshold Units

Threshold units contain the most complex processing. First, they perform a final sum over the partial sums from input sum processors. Then, they calculate the output after thresholding. The threshold output is $\frac{1}{1+e^{-\gamma S_j}}$, where S is the total sum and γ a gain which sets the threshold steepness. In the backpropagation direction, they compute the derivative of this function: $\frac{\gamma e^{-\gamma S}}{(1+e^{-\gamma S})^2}$. Since the ARM968 contains no native support for transcendental functions, computing sigmoids and their derivatives is most efficient by lookup table. However, 16-bit precision would require a 64k-entry table, larger than the DTCM size, so the units implement a fast spline-based interpolation [UVCP98] upon a 256-entry table.

5.3.2 Synapse

Synapses reside in the weight processors. These map a square [2] w_{ij} submatrix of the entire network, with outputs aggregating both in the forward direction along j components and in the backward direction along i components:

$$O_{fj} = \sum_i U_i w_{ij}, O_{bi} = \sum_j U_j w_{ji}$$

where O_{fj} and O_{bi} are the output of the weight unit to (network) unit j in the forward, i in the backward direction respectively, U_i and U_j the output of the final (threshold) processor for (network) unit i and j. This only requires transmission of order N rather than N^2 output values from both neurons i and j in each stage. The forward computation simply multiplies weight and input; using 16-bit values, the multiplication uses a single multiply-accumulate (the ARM968 SMLAxy instruction) per synapse. Given the trivial nature of the sum units' accumulate, it is reasonable to include the first stage of sums in the synapse unit as well, thus as soon as a given j "column" in the forward direction receives an input

[2]While it is possible to create synapse processors with rectangular aspect ratios this unnecessarily complicates the mapping problem, and therefore the implementation uses square matrices for convenience.

it adds the weighted value to its output sum. Once again this reduces traffic to the intermediate sum processors.The reverse computation must update the weights in addition to multiplying backpropagated errors by the weight. It first computes the new output, $O_i = w_{ji} * U'_j$. Next, it computes the new weight delta: $\delta_{ij}(p) = \lambda U_i U'_j + \mu \delta_{ij}(p-1)$, where $\delta_{ij}(p)$ is the change to weight w_{ij} for input presentation p, λ the learning rate, and μ a momentum term for momentum descent. The input U'_j from threshold unit j is the backpropagated error derivative. Lastly, it accumulates the new delta to the weight.

5.3.3 Dynamics

Having considered the mapping problem and identified appropriate methods to correspond neurons and synapses to processors, it is now possible to consider the process dynamics of the entire MLP implementation. Translating the MLP onto SpiNNaker involves converting the processing into an event stream. The formal MLP model is essentially synchronous: signals from the previous layer arrive at the next layer simultaneously for each pair of sources and targets. An event-driven asynchronous system like SpiNNaker cannot guarantee this level of input timing (indeed, it **forbids** it), and in any case, this would greatly slow down the processing. First, if all packets actually went out and arrived simultaneously, the system would have a very bursty traffic pattern liable to frequent transient congestion. Second, this would underuse the inherent parallelism of the architecture. Third and most fundamentally, the chip would have to signal the process clock as events. A pair of mechanisms: a packet format and an event definition, permit a completely event-driven MLP model on SpiNNaker.

Data Representation

SpiNNaker's AER packet format allows for a payload in addition to the address. In the model, therefore, the packet retains the use of the address to indicate source neuron ID and uses the payload to transmit that unit's activation. The result is an "address-plus-data" event: downstream processors in the compute-and-forward processing pipeline use the address field to determine which submatrix elements to update, and use the date in the payload to compute the update itself.

A few observations about the MLP mapping make it possible to determine a representation.

1. Whether a processor receives input or produces output to the external environment (it is a "visible" unit) or not (a "hidden" unit), it uses the same representation.

2. From the output of the weight unit to the input of the threshold unit there are no multiplication operations.

3. The output of the threshold unit in the forward direction is bounded between 0 and 1.

Combining observation 1 and 3, with respect to weight units, it is clear that a hidden weight unit operating in the forward direction computes the product of a weight and a number between 0 and 1. All other weight units use the same representation and so *external inputs must be mapped to the range 0-1.* This likewise scales external outputs by the same scaling factor.

Observation 2 implies that the maximum number of bits that intermediate computations could add is the input fan-in to the unit. Combining this with the fact that inputs are in the range 0-1, the largest (integer) value that an intermediate computation can reach is $(1 + log_2 F_{max})w_{ijmax}$, where F is the fan-in and the max subscript indicates the largest possible value.

Given studies indicating that 16-bit precision is adequate for most purposes [HF92], the weight unit's multiply operation implies that both the weight and the threshold unit's (forward) output should be (signed) 16-bit numbers, producing a 32-bit product. Since the second of these numbers is in the range 0-1, that intermediate result has its binary decimal point shifted left by 15 places. If the accumulated sum through the pipeline is not to overflow, the binary decimal point should thus be set so that $p = 15 - log_2(2w_{ijmax}MOD(1_l og_2 F_{max}))$ where p is the bit-position of the 1's place.

In practice it is it usually possible to improve the precision by placing p to the left of the restriction above, because in many networks very few weights ever reach w_{ijmax}, nor is it the case that the unit with the highest fan-in receives many inputs with this weight value. In such cases it is convenient to "saturate" sums, so that if a sum does exceed the maximum of the 32-bit representation, the processor clips it into the integer range 0x80000000-7FFFFFFF. The ARM processor contains carry/overflow instructions that perform this and can automatically set the saturation. There are therefore only 3 major internal representations:

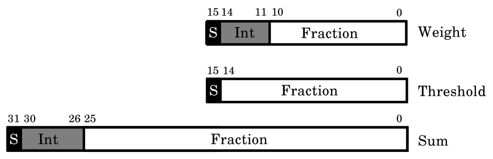


Figure 5.4: SpiNNaker MLP data representations

1. Weights are signed 16-bit quantities with an integer part and an implied fractional part. The application should determine the position of the binary decimal point to minimise the likelihood of overflow.

2. Threshold units output a signed 16-bit quantity with the binary decimal in bit 15 - "Q15" notation.

3. Intermediate sums between weight, sum, and threshold units are 32-bit signed numbers with the binary decimal point at $15 + p_w$ where p_w is the position of the weight representation's decimal point.

5.3.4 MLP event definition

There are two important events in the MLP model. One obvious event is the arrival of a single vector component at any given unit: "packet received". However, the dataflow and processing falls into 2 distinct phases: the forward pass, and the backward pass. Because the quasi-synchronous nature of the MLP ensures that a given unit will not output until it has received *all* inputs to it, it is safe to change the direction in that unit when it issues an output. Thus a second, internal event: "flip direction", triggers the change from forward to backpropagation on output sent.

In the SpiNNaker hardware, events map to interrupts. The communications controller generates a high-priority hardware interrupt when it receives a packet, while writing a packet (i.e. an output) to the communications controller triggers a software interrupt from which it is possible to reverse the direction. These 2 events preserve the characteristic of being *local*: a unit does not need to have a global view of system state in order to detect the event.

Packet Received

The packet received event drives most of the MLP processing. The processor uses the deferred-event model to handle incoming packets. Its device-level ISR, running as a maximum-priority FIQ, immediately pushes the packet data into a queue. It then triggers a deferred event process operating at system level to dequeue the packets and performs the relevant incremental computations. The background process handles the global computations necessary when all inputs for a given unit have arrived. Using the deferred-event model, the processor can therefore handle inbound packet servicing and intermediate processing as packets arrive, without having to wait for (and buffer) all packets.

To summarise the process in each unit, denoting the internal input variable as I and the output variable as J, exact processing depends upon the stage as follows:

1. Dequeue a packet containing a data payload.
2. Test the packet's source ID (address) against a scoreboard indicating which connections remain to be updated. If the connection needs updating,
   (a) For weight processors, $I = w_{ij}O_i$, where O_i the payload. For all others, $I = O_i$.
   (b) For weight processors in the backward pass only, compute the weight delta for learning.
   (c) Accumulate the output for neuron j: $J = J + I$
3. If *no* connections remain to be updated,
   (a) For threshold processors only, use a look-up table to compute a coarse sigmoid: $J = LUT(J)$ in the forward direction. Get the sigmoid derivative $LUT'(J^f)$ in the backward direction. (J^f is the *forward J*, J^b the *backward*.)
   (b) For threshold processors only, use the spline-based interpolation to improve precision of J.
   (c) For threshold processors in the backward pass only, multiply the new input by the derivative: $J = J^b(LUT'(J^f))$.
   (d) Output a new packet with payload J.

4. If the packet queue is not empty, return to the start of the loop.

Flip Direction

The unit sends the output for any given target neuron when all inputs for that target have arrived, and the communications controller transitions to empty. It does *not* wait for inputs destined for other targets the particular processor serves. Thus, if a weight processor forwarded to units X_1, X_2 and X_3, and it had received all inputs needed to compute the output for X_2, it would output that immediately rather than waiting for inputs for X_1. In order to issue an output (and change direction), therefore, the processor must detect that all inputs have arrived.

The most general way to do this is by a scoreboard, a bit-mapped representation of the arrival of packet-received events for each component. The test itself is then simple: XOR the scoreboard with a mask of expected components. While this method accurately detects both the needed condition and component errors, it has an important limitation: all inputs *must* arrive before the unit changes direction. "Fire-and-forget" signalling provides no delivery guarantees, so a receiving unit might *never* receive an expected input. This would effectively stop the simulation, because the network as a whole can proceed no faster than its slowest-to-output unit.

This "output-on-demand" model replaces the synchronous wave of computation with a flow of asynchronous signals between processes. Superficially, this alters the overall state of the network: in principle different parts of the network with different connectivity may be at a different point in the overall computation. The 2-pass nature of the backpropagation algorithm ensures that the overall result does not change. Under backpropagation, all terminal signals must arrive at the outputs before the algorithm can compute the error and reverse the direction; this is effectively a form of barrier synchronisation that prevents subgraphs from receiving inputs in the reverse direction until all forward computations have completed.

Output-on-demand also helps to spread traffic load, at least to the extent possible, in that individual outputs do not have to wait upon each other. However, because of SpiNNaker's asynchronous design, the communications fabric provides no packet scheduling or traffic management. This can lead to local congestion, with potentially severe impact on performance and in extreme cases, on functionality: if packets arrive at a given processor faster than it can service them then it will start to miss packets altogether. Processors issue packets as soon as they are ready: thus if several packets become ready to send simultaneously or nearly so there will be a "burst" of local traffic.

The "burstiness" of the traffic depends on the ratio of time spent in the packet-receiving service routines and the time spent in the background process from which the processor issues packets. If the time spent in packet reception is large compared to the time spent in the background task there is a strong risk of local congestion. This is precisely the situation in the sum processors, since the accumulate process takes only three instructions per update. To "stretch" the effective time in the background task so that it becomes large relative to the packet-processing task, sum processors can also assume the Monitor processor rôle, running the sum as an event-driven service and the monitor as the background task.

5.3.5 Network

Having completed process mappings and event definitions, it is possible to build MLP networks on SpiNNaker. The actual networks that have been built serve three testing purposes: basic functionality verification, scalability testing, and investigation of topological characteristics.

Basic Networks

Typically, multilayer perceptrons are feedforward networks with full connectivity between layers (fig 5.5). Thus the most basic model, and the network used for the first tests, is a simple full-connectivity feedforward network. The initial target application for the MLP is hardware acceleration for the LENS [3] software simulator. In this case, the initial SpiNNaker test network is a slight modification of LENS' digits-recognition network, containing 20 input neurons (or, more simply, inputs), 20 hidden neurons, and 4 output neurons. The model maps the network onto four SpiNNaker chips, using one processor to model the threshold, and three apiece for sum and weight units. The digits-recognition network specifies neurons with sigmoid transfer function using the form from 5.3.1. Learning is per Lens' momentum rule given in 5.3.2. For the test network, λ was 0.0078125 and μ was 0.875. The batch size was set to 1, corresponding to updating the weights after

[3]http://tedlab.mit.edu/~dr/lens

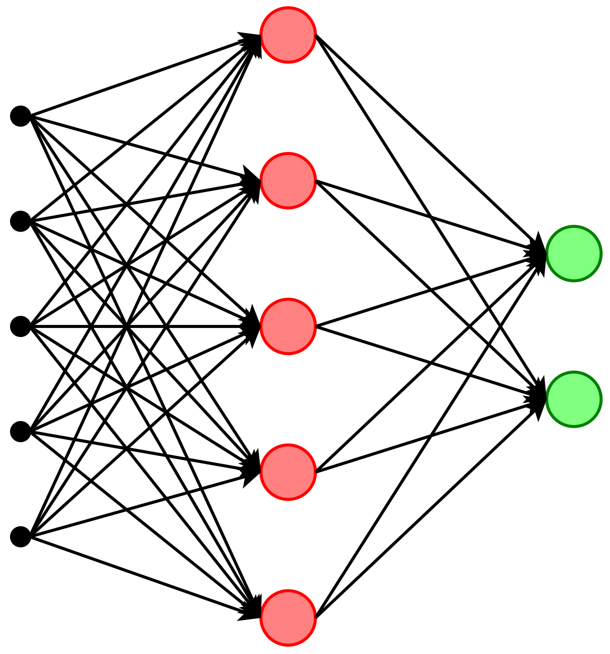


Figure 5.5: Typical MLP network with full connectivity. The first test network has this structure with expanded numbers of units per layer.

each presentation of an example. For the training (example) set, the test augments Lens' standard digits examples (containing digits 0 through 7 with digits 8 and 9, and supplements these with two sets of distorted digits: fig. 5.6. Training stops after processing 120 examples, corresponding to 3 complete epochs.

Scalability testing

Scalability of the MLP model is an important concern with the unusual mapping used for SpiNNaker, especially in view of the packet-traffic issues relating to burstiness. It is important to know how far the SpiNNaker architecture can scale

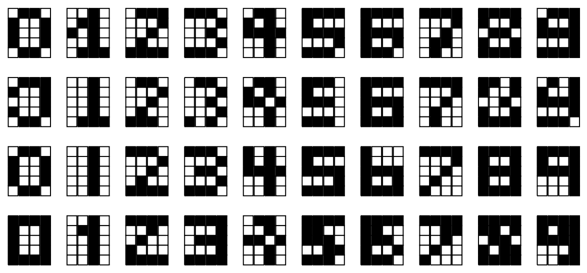


Figure 5.6: Test inputs used with MLP networks

in terms of local packet density before breaking down. A simple network to test packet-handling performance artificially increased the packet traffic by extending the digits-recognition model, generating additional packets from the actual traffic of the simulation proper. These networks generated a separate packet type that output the weight of each synapse, for each input. A further network parameter specifies the number of copies of the packet the processor issues per weight update, thus enabling an arbitrarily high packet traffic density.

Such artificial techniques are useful to understand basic limits and establish ultimate performance, but they do not represent a "realistic" scalability test: the system does not scale the number of neurons or synapses, important for memory testing. In addition, the artificial "debug" packets the previous network uses run over a fixed route, and therefore do not test *overall* communications capacity or the effects of distributing high packet traffic throughout the network. Therefore, to create a more realistic case, the tests use a larger network designed to recognise spoken words from a small fixed vocabulary (a development of the model in [WR07]) with further assistance in the form of visual input of the characters. This network contains a single input layer and 2 distinct groups each of hidden and output layers representing the visual and sonic characteristics.The input layer contains 400 units. The 2 output groups contain 800 and 61 neurons respectively. The 2 hidden groups have variable numbers of neurons; from 50 to 200 neurons per group. All groups use the standard sigmoid function in 5.3.1 for their thresholds. Inputs connect only to hidden group 1 from which there are 2 main connectivity paths: to output group 1 and to hidden group 2. Hidden group 2 then connects to output group 2 to establish a forking pattern of connectivity. In addition there are further connections from the input directly through to output group 1, and from output group 1 itself into hidden group 2 and output group 2 (fig. 5.7). Because of its larger size, this network is a good test for alternate topologies as well as scalability.

Partial Connectivity

Full connectivity is adequate for small networks, containing up to $\sim$1000 neurons, but computationally unmanageable at larger scales. Efficient network connectivity becomes particularly significant in the context of hardware systems. In such devices, available circuit densities constrain the number of connections available

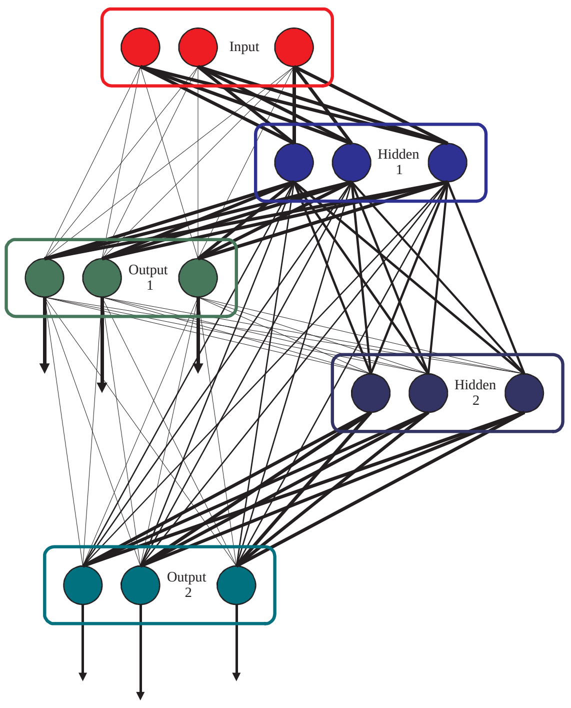


Figure 5.7: Large phonetic network used in tests

at any given size. For SpiNNaker in particular, the 1024 entries in the local routing table tend to be the limiting factor establishing routability of the network [KLP+08]. It is also essential to balance the communications load across the network. SpiNNaker's routing model assumes average-case traffic conditions with about 10% load; full connectivity will magnify the potential for traffic bursts already noted, especially in large systems, because all sums (and weights) in a given processor will become ready to send as soon as the last partial update arrives. Partial connectivity mitigates this problem since different matrix subelements become ready to send at different times.

It is unreasonable to assume that random partial connectivity will lead to an optimal structure in a large network performing a specific task; it might rather be expected that the task itself would suggest a structured pattern of connectivity matching in some way the characteristics of the task. If, however, by reducing the number of connections important performance measures like training time or classification speed deteriorate, then the network optimisation is of little value. Therefore the goal is to find connectivity patterns that maximise the speed and learning rate per connection.

Previous connectivity examinations have primarily focussed on evaluating the effects of varying connectivities in a network with a fixed number of *neurons* [DCA06], [CAD07]. In a hardware system such as SpiNNaker, however, it is more likely that large neural networks will be connection-limited than they are neuron-limited, given the finite routing resources. Simply varying connectivity with a fixed number of neurons may not lead to an optimal hardware-implementable network. With a connection-limited system, it is practical to vary the number of neurons since they do not add substantially to the resource requirements. Since in the MLP model, both the number of input neurons and the number of output neurons remain fixed, because of the need to present the network with external input and training class identifications, only the hidden neurons may vary in number. By adopting a model where the number of *connections* is fixed, and the number of (hidden) neurons (and hence the mean connectivity per neuron) varies, it is possible to examine the impact of partial connectivity in a more hardware-realistic way.

For the first set of tests, a simple network implements Lens' hand-digits network: a relatively logical extension of the basic digits recognition task. This network contains 64 input, 10-40 hidden, and 10 output neurons in a simple feed-forward configuration. The tests vary the number of hidden neurons and connectivity ratios from input-hidden and hidden-output in 5 steps of 20%. Hidden neurons use the sigmoid transfer function from 5.3.1. Outputs use the SOFT_MAX function, which computes $e^{\gamma S_j}$, where γ is the gain, as earlier, and S_j is the input to the neuron, normalised so that the sum of contributions from all of the outputs sum to 1. The tests also vary the gain (normally 1) in all layers in decreasing steps of 0.2: 1, 0.8, 0.6... Learning is per Lens' "Doug's momentum" rule which adjusts the standard momentum rule $\delta_{ij}(p) = \lambda U_i U'_j + \mu\delta_{ij}(p-1)$ such that $delta_{ij}(p-1)$, the previous weight update, is set to 1 if it was larger than 1. In this network, λ was 0.2 and μ was 0.9. The batch size uses the entire example set, i.e. the network does not update the weight until after a complete presentation of the example set. The training set uses Lens' supplied hand digits examples from hand-digits.trn.bex.gz. Training stops either when all output neurons have an error less than 1, or when the network has gone through 100 complete presentations of the training set without learning the task. The network then uses Lens' supplied test set: hand-digits.tst.bex.gz, for connectivity/gain parameter combinations that successfully learn the task.

Smaller networks, however, can only reveal effects at a coarse level, because the relatively small number of connections limits the number of useful topological variations. The large phonetic-recognition network is an obvious candidate for testing partial connectivity with finer control over the topology parameters. Tests allowed the size of the hidden layers to vary between 50 and 200 neurons (keeping the number of input and output neurons the same). The tests vary the connectivity in all hidden layers in 3 decreasing steps: 1, 0.66, 0.33. Again, learning follows Lens' "Doug's momentum" rule. with $\lambda = 0.05$ and $\mu = 0.9$. Batch size was 1, specifying a weight update following presentation of each example. Both training and test sets used a phonetic example set from the School of Psychological Science, University of Manchester, "namtr.ex", consisting of 2000 vocalised phonemes, dividing the set evenly into two groups of 1000 for training and test respectively. Training stops either when both output groups have a total error less than 0.5, or when the network has gone through 1000 complete presentations of the training set without successfully learning the task (i.e. meeting the criterion). These networks permit a systematic exploration of the relationship between connectivity, network size, gain, and learning performance.

The connectivity-test networks complete the entire model development cycle on SpiNNaker; with them testing has gone from investigating the functionality of the library model to testing characteristics of the networks itself. This is the task SpiNNaker was designed to do; thus the MLP networks can demonstrate SpiNNaker's value in non-spiking networks in addition to spiking ones. Nonetheless, *spiking* networks are currently the ones attracting the greatest interest in the research community, and to demonstrate how SpiNNaker library techniques permit rich exploration of network characteristics in the spiking case, it is instructive to turn to a third model.

5.4 The LIF/NMDA Model

The purpose of SpiNNaker is large-scale simulation. While simple models like the reinforcement-learning subcircuits demonstrate basic functionality and core techniques for event-driven neural implementation, they do not as yet form convincing evidence of scalability. Models such as the MLP are useful in demonstrating medium-scale neural networks, and in addition the very different nature of the underlying model readily exposes potential hardware limitations. Again, they are only partly convincing as a demonstration of scalability, in part because the target network sizes are still only relatively small, in part because as a "non-standard" model they use SpiNNaker in a very different way from the original design objectives. In particular, however, they are not dynamic models: there is no real-time requirement and no temporal dimension. These models remain *illustrative* rather than *prescriptive*; having developed them it is now important to introduce further models to build a neural library and provide further examples to third-party developers of how to configure SpiNNaker. The third set of networks focusses on an important, popular neural model - the leaky-integrate-and-fire (LIF) model, and uses it as a platform to develop other synaptic models, notably an N-Methyl-D-Aspartic (NMDA) synapse with slow voltage-gated dynamics. These models introduce a variety of techniques not yet demonstrated in the original reference models, while using and extending important core techniques from those models that show their general nature. Developing different neural models is a useful way not only to extend hardware capabilities, but to establish principles of efficient computation - a basis for approaching the question of what abstractions of neural function are useful.

These "neural primitives" are designed to integrate into a real-time, spiking neural network as part of a complex model operating in a biologically relevant time-sensitive application: attentional control. The goal of the system is to be able to recognise and direct (visual) attention towards a set of prioritised synthetic stimuli moving over a virtual visual field. This requires coordinated interaction between a set of distinct functional modules, each a neural network by itself. The obvious structural and dynamic complexities of such a model make it strongly preferable to use simple neural models.

5.4.1 Neuron

The LIF model uses the function pipeline common to the SpiNNaker library. This basic approach applies to virtually any spiking model with voltage-variable differential-equation dynamics: an illustration of the universal design of the software as well as the hardware. It is perhaps easiest to describe the LIF neuron by walking through the function pipeline stage by stage.

Variable Retrieval

Like the Izhikevich neuron, the LIF neuron uses the deferred-event model to place input spikes into a circular array of current buffers representing the total input in a given time step (fig. 5.1). At the actual time of arrival of an input, the model does nothing more than trigger a DMA operation which retrieves a source neuron-indexed synaptic row. Once this row has been retrieved, a deferred event, DMA complete, triggers a second stage of deferral by finding the delay associated with a given synapse and placing the synaptic weight (representing the current injection) into the bin in the circular array representing the delay value. Finally, when that delay expires, it triggers another deferred event, the Timer event, that recovers the total current from the bin. At the same time, the neuron retrieves its associated state block, containing the voltage variable and the parameters V_r (rest voltage), V_t (threshold voltage), V_s (reset voltage), R (membrane resistance), and $f_n(=\frac{1}{\tau})$ (natural frequency). Precomputing this last value from time constant τ makes it possible to avoid an inefficient, processor-intensive division.

Polynomial Evaluation

The basic LIF neuron equation is [DA01]

$$\frac{dV}{dt} = f_n(V_r - V + IR)$$

The right-hand side is a simple polynomial, which the ARM can easily compute (in 3 instructions using 2 SMLAxx multiply-accumulates). It is possible also to incorporate conductance-based synapses into this evaluation; the function then becomes

$$\frac{dV}{dt} = f_n(V_r - V + T(t)(V - V_n) + IR)$$

where T(t) is the synaptic "transmissivity", a value that incorporates the values of maximum synaptic conductance and specific membrane resistivity into the release probability by precomputing and storing the weights as this aggregate quantity. V_n is the synaptic rest voltage.

Look-up table

For the "normal" LIF neuron there is no need for a lookup table function since the differential equation is polynomial.

Interpolation

Likewise in the standard LIF model there is no need for interpolation since the function can be computed exactly.

Differential Equation Solution

The Euler-method process evaluates the differential equation at each Timer event. Timer events occur each millisecond. After evaluating the equation, it also tests whether the potential has exceeded the threshold V_t. If it has, it resets V to V_s.

The LIF computation requires 10 instructions if the neuron does not fire, 21 if the neuron fires, and 38 if the neuron fires and synapses have spike-timing-dependent plasticity (STDP). The high efficiency of this model makes it an ideal test-bed for exploring different synaptic models.

5.4.2 Synapse

Spiking neural networks can contain various different synapse types with different dynamics. At present the role of different synaptic types remains an area of intense research interest [Dur09]. Equally significantly, the level of biophysical realism necessary to achieve useful behaviour or model actual brain dynamics is unclear. For instance, in the case of the well-known STDP plasticity rule, while many models exist describing the behaviour [SBC02], [HTT06], the actual biological data on STDP is noisy and of low accuracy. Observed STDP modifications exhibit a broad distribution for which the nominal functional form of STDP models usually constitute an envelope or upper bound to the modification [MT96], [BP98]. This suggests that high repeatability or precision in STDP models is not particularly important.

While SpiNNaker is capable of modelling synapses with biophysical realism down to the molecular level if necessary, such high biological fidelity is computationally expensive. In view of the observed synaptic variability, exact biological replication, or fidelity to a precise functional form, appears to be unnecessary for understanding their computational properties. This gives considerable latitude for experimenting with different synaptic models in order to investigate various tradeoffs between computational cost and functional accuracy. Using the LIF model gives not only a default “reference” model with known and easily tuned dynamics that expose the synaptic dynamics clearly, it also provides a very-low-computational-cost model, minimising machine cycles to allow more time for complex synaptic models.

The LIF model currently supports two different models. The first model is the temporally-accurate STDP model from earlier 5.2.2. In biological terms, this model implement synapses with fast dynamic response, either AMPA (excitatory) or GABA-A (inhibitory) types The second model adds NMDA-mediated synapses exhibiting voltage gating with slow dynamics. This model presents two challenges: first, how to model the slow dynamics without having to retrieve the synaptic values multiple times, and second, how to implement the voltage gating process. Two properties of NMDA synapses make it possible to implement an efficient algorithm. First, the decay of the current exhibits a *linear* kinetic, therefore instead of using sampled Euler-method evaluation it is possible to precompute the update for each input. Second, voltage gating is multiplicative and depends on the *post-synaptic* potential, thus the net contribution from all synapses with NMDA-mediated dynamics can be computed by multiplying their aggregate activation by the gating factor on a per (postsynaptic) neuron basis.

Variable Retrieval

Retrieval of the synaptic weight follows the same virtual synaptic channel method as the "ordinary" AMPA/GABA-A synapse: the processor initiates a DMA transfer upon receipt of an input spike. However, since at this time the type of synapse (NMDA/AMPA/GABA-A) is unknown, each synaptic weight entry uses the previously-reserved Bit 27 to identify the type. A type 1 indicates a slow dynamic; the sign bit of the weight whether the synapse is excitatory or inhibitory. Thus an NMDA synapse has a positive weight value and a 1 in Bit 27. The processor then branches to the appropriate routine depending on the synaptic type.

Polynomial Evaluation

The full equation for the NMDA synapse is a term in the neuron that specifies the amount of current injection [DA01]:

$$\left.\begin{aligned} I_N &= r_\nu g_n G_n P(V - E_n) \\ G_n = \frac{1}{1 + \frac{Mg^{2+}}{3.57} e^{\frac{-V}{16.13}}} \quad \frac{dP}{dt} &= \alpha(1 - P) - \beta P \end{aligned}\right\}$$

r_ν is the neuron's specific resistance, g_n the NMDA synapse conductance, P the open probability, and E_n is the synapse rest potential. G_n is the voltage gating factor; it depends on the magnesium ion concentration Mg^{2+}. Factors α and β are (approximately constant) factors that determine the synapse opening and closing time constants. Precomputing the synapse open probability uses a variety of techniques. NMDA synaptic currents reach their maximum value quickly, usually within 1.5 ms of spike arrival, then decay exponentially with slow time constant β :

$$P(t) = P_{max} e^{-\beta(t - T_{max})}$$

where P is the synapse open probability, P_{max} the maximum "fully-open" probability, and T_{max} the time of that fully-open state. Since the rise time is fast, of the order of the time resolution of the neural dynamics, it is reasonable to neglect the effects of rise-time dynamics and simply incorporate it into the delay. Also, the factors r_ν and g_n are constants. This makes it convenient to bundle them into the "weights" in the stored NMDA synapse. The net effect is that this "weight" is actually a term equivalent to $r_\nu g_n P_{max}$.

With these numerical manipulations, the actual evaluation of the NMDA expression is straightforward: compute $V - E_n$, and multiply by the stored "weight". The gating factor G_n however, is a nonlinear sigmoid 5.8, thus it is easier to obtain this by a look-up table.

Look-up Table

Using a look-up table for the gating factor "collapses" the entire evaluation into a memory access, however, the Mg^{2+} term has the net effect of shifting the actual sigmoid to higher voltages with higher ion concentrations 5.8. This is easy to achieve by changing the voltage which gives the offset into the lookup table by a factor proportional to the Mg^{2+} term. Again, it is easier to precompute this

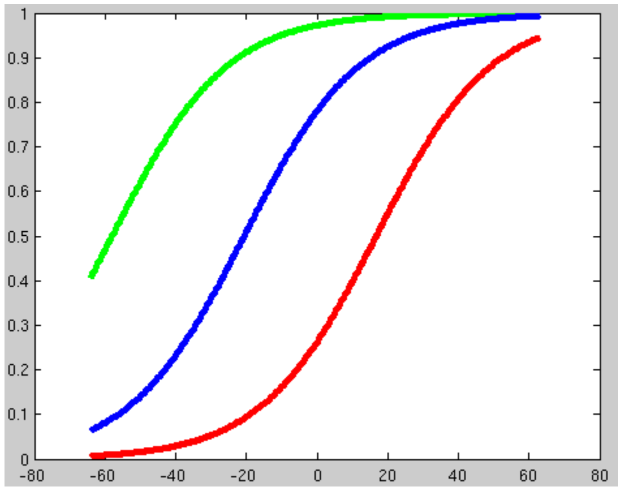


Figure 5.8: NMDA gating factor as a function of membrane potential. Traces represent different values of Mg^{2+}: 0.1mM (green), 1 mM (blue) and 10 mM (red)

actual offset (and bundle in the divisons by 3.57 and 16.13) so that is is only necessary to add a single value stored in memory representing the effective Mg^{2+} concentration in order to get the LUT offset.

Interpolation

If the model uses NMDA synapses, and requires better than 8-bit precision in the gating factor, it uses the same spline-based interpolation [UVCP98] as the MLP model. Since the LIF neuron and NMDA synapse are dynamic models running in real time, as opposed to the timeless MLP model, in practice it is best to avoid this step unless absolutely necessary: while efficient, interpolation is not computationally free, requiring 48 cycles.

Differential Equation Solution

Because the NMDA synapse has a linear kinetic, it is more efficient to precompute its net input to the neuron rather than follow Euler-method integration. NMDA input current decays exponentially with a slow time constant: $\beta \gg \tau$ where τ is the time constant for a fast AMPA synapse. It is straightforward to accommodate this slower dynamic by adding a second set of activation bins to each neuron that rotate every τ ms. Precomputing the value to place in each successive bin after the initial (T_{max}) bin is trivial by exploiting the exponential property of shifting and using precomputed parameter scaling: shift the “weight” stored in the synaptic block, right by one place for each successive bin.

At each (1ms) time step, the neuron then multiplies the current NMDA bin (representing aggregate NMDA stimulus) by the voltage gating factor and adds it to that of the normal fast (AMPA/GABA-A) bin using the previously-described LUT-based scheme. Once again this function can exploit extensive precomputation to simplify the LUT evaluation. The neuron also needs to test whether the NMDA bin should rotate, based on the system time t_{sys}. Naïvely, this is when $t_{sys} REM \tau = 0$. However, because of the finite time resolution of the system, it should actually occur when $t_{sys} REM \tau \leq \frac{1}{2\tau}$. By storing the time constant as a 16-bit fractional quantity with implied leading decimal, $f_{nmda} = \frac{1}{\tau}$, determining the rotate timing requires a simple multiply and shift: $(t_{sys} f_{nmda}) \ll 16 \leq f_{nmda} \ll 15$.

The complete process then functions as follows:

Presynaptic input event

1. Retrieve the synaptic row corresponding to the presynaptic neuron.
2. Calculate the effective bin delay (position of bin P_{max}) using $T_{max} = \delta_w(\frac{1}{\tau})$, where δ_w is the real-time annotated delay in the synaptic weight.
3. Precompute the synaptic contribution (effective open probability) for each future bin.
4. Distribute the contributions into each bin by accumulating them with the current bin value.

Neuron timer event

1. Get the current system time.
2. Calculate whether the NMDA bin should rotate by computing $t_{sys} REM \tau \leq \frac{1}{2\tau}$.
3. If the bin should rotate, advance it and clear the previous bin.
4. Compute the gating factor by LUT.
5. Multiply the current NMDA bin by the gating factor.
6. Add to the (fast AMPA/GABA-A) activation.

The NMDA synaptic model takes advantage of the low instruction count of the LIF process. Updating the NMDA dynamics in a neuron requires 22 instructions. Thus the total instruction count rises only to 32 if the neuron does not fire, 43 if the neuron fires. This remains well within timing limits to permit a (memory-limited) 910 neurons per processor.

5.4.3 Network

The Large-Scale Model

Because of the underlying simplicity of the component elements, this model is suitable for implementation of a larger-scale network with a more complex structure than the previous models. This network is a good demonstration of a scalable system containing heterogeneous components: a "full-scale" network to investigate realistic behaviour in a real-world environment.

The proposed network (fig. 5.9 is an expansion of a pair of models: Taylor, et. al's CODAM model [Tay03] and Edelman, et. al's Darwin V model [STE00]. Darwin V's S, S_o and S_i units replace the monitor and goals subunits in CODAM. As planned, the network will map S_o and S_i to the Evaluator (the test network for the Izhikevich/STDP model), and S to the Monitor. When fully implemented, the network will also incorporate a long-term memory not in any of the previous models. Using a long-term memory will make it possible for the network to supply attentional priming from learned patterns, thus being able to precondition faster response to familiar stimuli in familiar contexts.

Subcomponents

Within the model there are several structures that repeat in multiple independent function blocks. Following the libary modular structure, the first step is to build and test these blocks as independent networks, then incorporate them into the larger system. Such a modular approach provides the benefits of block reuse and incremental debugging, so that large-scale neural networks possessing repeated regular structures can be built by connecting components without the risk of problems in the lower-level subcomponents obscuring top-level behaviour.

The first such structure is a "comparator" that evaluates the synchronisation of paired inputs and outputs a coded pattern representing the phase delay between inputs. The test implementation reproduces the results in [IH09] (Section 2).

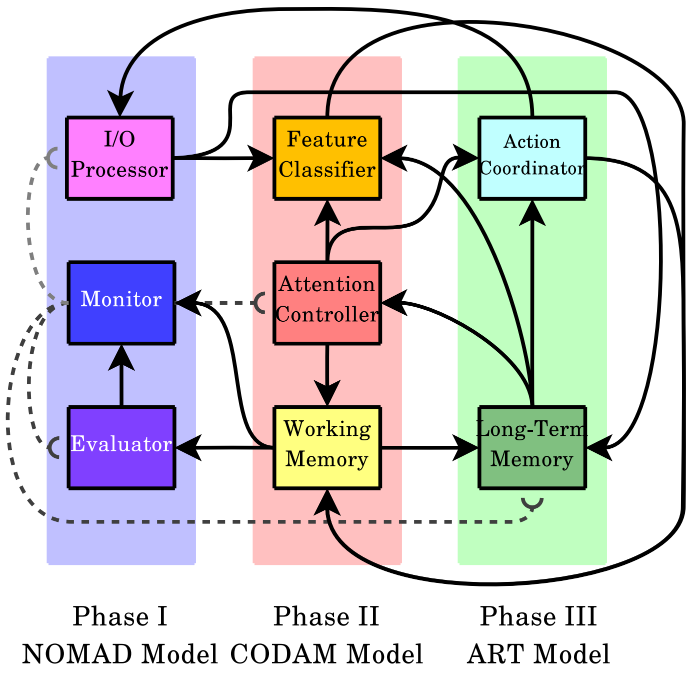


Figure 5.9: Proposed full-scale model for scalability testing. Each block is a separate network and may employ independent neural model types. The Evaluator uses Izhikevich neurons with STDP and is the subject of section 5.2.3. The Monitor uses LIF neurons; its core will be a resonant network containing LIF neurons with NMDA synapse. The I/O processor is a simple first-level feature extractor; the Phase I implementation will connect I/O directly to the Monitor and Evaluator, using synthetic input with fixed-activity neurons (Future implementations will incorporate real-world stimulus sources and outputs). Other blocks (Phase II and Phase III) are not yet implemented.

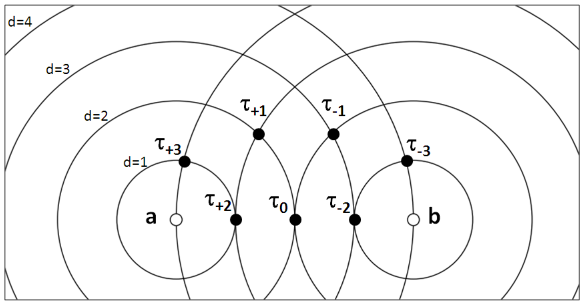


Figure 5.10: Detection of the interpulse interval between spikes from neurons a and b. d denotes the synaptic delay between sources and detectors. Detector neuron τ_i fires when the interpulse interval $t_a - t_b = i$.

Two input neurons (sources) connect to 7 output neurons (detectors) with a delay proportional to the distance (fig. 5.10).

The weights are set so that a detector will fire only if two coincident spikes arrive within a millisecond interval; detector neuron τ_i only fires when the interpulse interval $t_a - t_b = i$, where t_a and t_b are the absolute firing times of neurons a and b respectively (eg. neuron τ_{-3} fires when neuron b fires 3 msec *after* neuron a, neuron τ_{+2} fires when neuron b fires 2 msec *before* neuron a etc). This yielded a uniform weight of 2.7 mA/mF with delays of 1, 1, 2, 2, 3, 3, and 4 ms respectively depending upon the detector neurons's distance from the source neuron.

Source neurons receive a regular series of 10 mA/mF current pulses, occurring at $\sim$ 20 ms intervals with a decrementing 3-0 ms offset for one source and an incrementing 0-3 ms offset for the other. Decrementing offset starts after 20 ms and diminishes by 1 ms every 20 ms, while the incrementing offset starts after 80 ms and increases by 1 ms per 20 ms. The simulations performed stopped input automatically after 200 ms.

The second test network is an oscillatory network (fig. 5.11) consisting of groupings of excitatory/inhibitory neurons. Such a network is useful either to generate a constant "clock" or in combination with a winner-take-all network to produce networks able rapidly to switch between sources of attention based upon the oscillation frequency. The model contains 100 neurons divided into 80 excitatory neurons and 20 inhibitory neurons. Each neuron in the excitatory group connects to 56 (70%) excitatory neurons and 2 (10%) inhibitory neurons with a random delay between 1 and 8 ms. Each inhibitory neuron connects to every excitatory neuron (full connection - 100%) with a delay of 1 or 2 ms. All excitatory weights have uniform starting weight 2 mA/mF, while inhibitory weights likewise

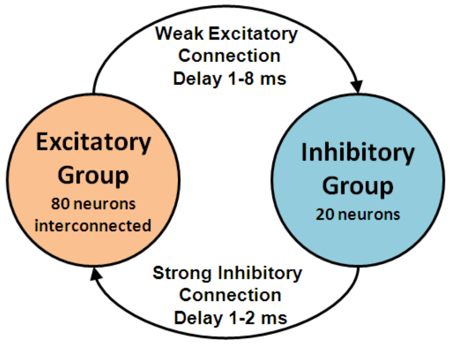


Figure 5.11: Oscillatory Network Structure. Excitatory and inhibitory groups are connected so that when the activity of the excitatory group gets high the inhibitory group shuts the activity of the whole network.

have uniform starting weight -20 mA/mF. 8 neurons in the excitatory population receive a constant current injection of 3 mA/mF starting at 10 ms and ending at 500 ms. The simulations stopped at 501 ms.

To test network dynamics at a larger scale, a larger network simulates 500 LIF neurons running the "Vogels-Abbott" benchmark. This creates a network similar to the 100-neuron oscillatory network. 400 neurons are excitatory, 100 inhibitory, and the neurons all have the following parameters: $V_i = -60mV, V_r = -49mV, V_s = -60mV, V_t = -50mV, \tau = 32ms$. The population uses random connectivity with 2% probability of connection, i.e. a given neuron connects on average to 10 other neurons. Weights for excitatory connections are a uniform 0.25 mA/mF and -2/25 mA/mF with random delays between 1 and 14 ms. A subset of 40 neurons receives a current input injection; this current increases in 2 step-jumps at 400 and 800 ms. Simulation stops at 1200 ms.

In addition to networks using pure LIF neurons, two additional models run multimodel simulations containing heterogeneous mixes of neural types. These models verify, then implement, and important part of attentional control: the generation of gating signals to output populations. In particular, as planned, burst signals from the Evaluator will gate LIF neurons in the Monitor. The models propose that burst signalling can act as a gating source to LIF neurons by rapidly pumping the membrane potential up to a near-spiking regime. Since LIF neurons cannot generate spike bursts, the model uses Izhikevich neurons as the burst souce.

The first network uses two layers of 15 neurons each, one layer of LIF and one of Izhikevich "Intrinsic Bursting" neurons. Each neuron in a layer connects with a single neuron in the opposing layer with reciprocal connections having random delays between 1 and 10 ms. Weights from Izhikevich to LIF neurons have a uniform initial value of 5 mA/mF, while those from LIF to Izhikevich have a value of 0.01 mA/mF. Six neurons in the Izhikevich population receive an input stimulus in the form of a stepped current source. This source alternates between 0 and 20 mA/mF. Two neurons receive step-ON times of 50 and 666 ms with step-OFF times of 500 and 1000 ms, two have ON times of 150 and 500 ms, OFF times of 300 and 750 ms, and two ON times of 175 and 525 ms, OFF times of 325 and 775 ms. The simulations ran for 1 s.

The second network implements the burst gating subcomponent for the Evaluator, configuring 3 layers of 15 LIF, Izhikevich "Chattering" and LIF neurons respectively. As in the previous model, one pair of LIF and Izhikevich neurons have mutual one-to-one connections with random delays from 1 to 11 ms in this case, and with weights of 4 mA/mF from Izhikevich to LIF, 0.01 mA/mF from LIF to Izhikevich. The third population, of LIF neurons, connects to the other LIF population with one-to-one connections having identical weight and delay properties as from the Izhikevich population to the first LIF population. Two Izhikevich neurons receive an input stimulus of 20 mA/mF starting at 200 ms. The LIF neurons in the third population are configured with a rest potential above threshold, so that after spiking they relax back to another spike with a constant frequency of 50 Hz. Simulation stops after 2 s.

The complete system will be a demonstration of important library techniques for scalability. Repeated reuse of primitive subcomponents to create larger-scale networks indicates how to use the library for synthesis: hierarchically building components that a synthesis tool can then lay down as drop-in blocks. Integrating different neural types within the same overall system demonstrates SpiNNaker's ability to support simultaneous interacting heterogeneous models. Finally, implementation of simplified network models believed to represent "real" biological regions or processes suggest a path to full-scale biological modelling: expand the same architectures with larger block sizes and denser connectivities, an incremental process where it is possible to verify the biological correspondence at each successively greater level of detail by direct SpiNNaker simulation. The model remains partial at this point but nonetheless provides important test results that establish practical procedures for building an event-driven library while revealing potential hardware limitations.

5.5 Neural Model Summary

Each of the three different reference models illustrates different aspects of SpiNNaker design tradeoffs. Undoubtedly these reveal that choice of model implementation on any neuromimetic system is highly platform-specific, placing a further urgency on model libraries so that users do not have to develop neural models from scratch for the hardware. The ability to configure and run very different neural models on the same hardware, however, is a powerful (and *essential*) feature. Most importantly, development of these three models establishes a reference methodology for library development, so that future models may be built using the same approach with maximal reuse of existing software components. To summarise the three models:

Izhikevich/STDP
: The Izhikevich neuron is a simple model that yet replicates all observed neural behaviours including bursting, while the STDP synapse is by far the most popular model for weight update in spiking networks. These models illustrate the main challenges in creating a reasonably biologically plausible spiking model: dynamic fidelity, memory utilisation, and instruction efficiency in the context of a "use-it-or-lose-it" event-driven system. The principal solutions to these problems are the deferred-event model, the virtual synapse channel, and the assembly "software macro".

MLP/Delta-Rule
: The MLP model, or specifically the sigmoid-unit neuron with delta-rule synapses, is the classical "computational" neural network whose biological realism is at best analogous. Given its timeless, continuous-activation representation it unsurprisingly introduces completely different design challenges, the most significant of which are: network mapping, communications traffic management, and barrier synchronisation. The mapping problem is solvable using an adroit choice of matrix subranging [JLK$^+$10] but the other two are at best mitigated with careful communications distribution techniques.

LIF/NMDA
: The LIF is the simplest widely accepted spiking model, making it a good test-bed for complex synaptic models such as the NMDA model where aggressively efficient neural processing is essential. Such models create a

third series of challenges: parameter specification, complexity tradeoff, and network-level dynamics. Solving the problem of parameter specification - that different model “flavours” may have different numbers of parameters, is a matter of generalising the System Level library code using templates. The other two, however remain open issues at least in part: pre-instantiation simulation can determine whether model complexities may cripple performance, but on the whole this remains a matter of hand-tuning.

Chapter 6

Phase II: Simulation - Modelling

6.1 Guide

An essential component of the model design process is software testing and simulation, prior to implementing library components and neural networks on the actual hardware. In the case of SpiNNaker, furthermore, this testing has the additional purpose of verifying the hardware design - since a hardware-level simulation of a complete running network is the best, indeed, possibly the only feasible, way of adequately verifying a large, complex hardware component. If working hardware is available the main purpose of software simulation is to confirm the viability of the neural model and the fidelity of its mapping onto the device(s). This testing and verification process has two stages: abstract modelling and hardware simulation. Abstract modelling occurs at the Model level and confirms the basic dynamics of components or the entire model as well as establishing a "reference behaviour" to compare against the physical implementation. Hardware simulation happens at the System level and verifies that the hardware successfully implements the model, as well as helping with debugging. This chapter discusses test results from the 3 different models using this structure. Each section, ordered by model type: Izhikevich/STDP, sigmoid/delta function MLP, and LIF/NMDA, reports results first from high-level modelling using a variety of typical neural network tools and then hardware-level simulation using the ARM SoC Designer SystemC simulator. Here hardware-level simulation means tests on standard PC's simulating SpiNNaker hardware, not actual simulation on SpiNNaker chips (the subject of the next chapter). There are two different types of test: functionality testing, which simply verifies whether the model can run at all; and performance testing, which seeks to establish expected SpiNNaker capabilities. Summarising the results, functionality testing shows that the models can indeed be made to run, but while performance testing reveals the potential for significant acceleration it also shows the critical impact of network traffic and the need to manage this carefully in order to avoid slowdowns.

6.2 Izhikevich/STDP Issues: Realistic STDP, Accurate Dynamics

6.2.1 Voltage-Domain Performance

The Izhikevich model with STDP synapses provides an early platform to develop and test applications for SpiNNaker while the hardware is still in the design phase. While it is possible to verify the cycle accurate behaviour of individual components using HDL simulation, verifying the functional behaviour of a neural computing system on the scale of SpiNNaker would be unmanageably complex, either to demonstrate theoretically, or to simulate using classical hardware description languages such as VHDL and Verilog. Therefore, a separate part of the SpiNNaker project created a SystemC system-level model for the SpiNNaker computing system to verify its functional behaviour - especially the new communications infrastructure. SystemC supports a higher level of timing abstraction: Transaction Level Modelling (TLM), exhibiting "cycle approximate" behaviour. In a TLM simulation, data-flow timing remains accurate without requiring accuracy at the level of individual signals. This makes it possible on the one hand to integrate synchronous and asynchronous components without generating misleading simulation results, and on the other to retain timing fidelity to the neural model since the data-flow timing entirely determines its behaviour. The SpiNNaker simulator integrates bespoke SystemC models for in-house components [KJFP07] with a cycle-accurate instruction set simulator for the ARM968E-S processor and its associated peripherals using ARM SoC Designer. SoC Designer does not support real-time delays, therefore the model describes the behaviour of the asynchronous NoC in terms of processor clock cycles. The model simulates 2 processing cores per chip - the number of cores on the initial test chip - in a system containing 4 chips running concurrently - the configuration of the test board. Two neural application case studies [KLP$^+$08] tested the complete system behaviour extensively to verify hardware functionality before running model-specific tests. The complete process, from device-level functional verification to model-level simulation demonstrates the viability of the SpiNNaker platform for real-world neural applications and establishes the methodology for the development and testing of new neural models prior to their instantiation in hardware.

Low-level system tests initially ran on a system simulation containing one

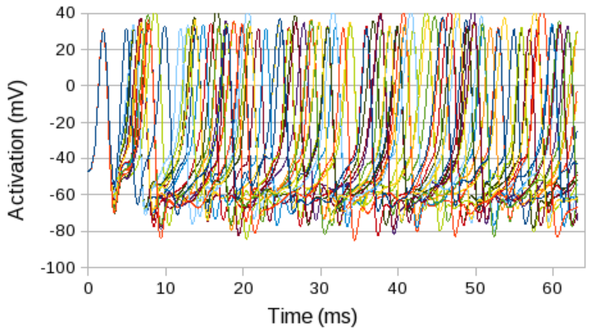
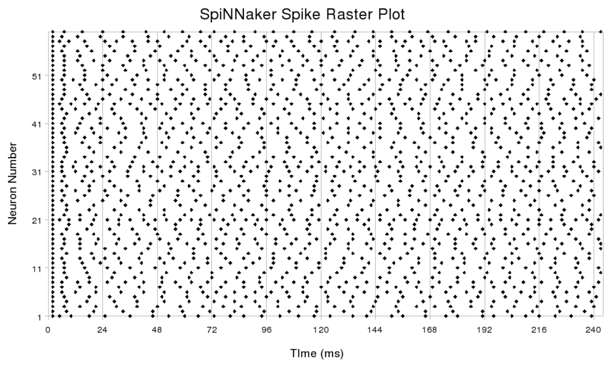


Figure 6.1: SpiNNaker top-level model output of the spiking network. For clarity only 25 neurons are shown.

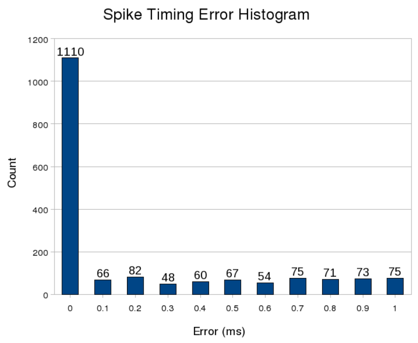
processing node. Router links wrap around connecting outputs to inputs, so that all packets go to the emulated processor. These tests (Xin Jin) used a reference neural network having 1000 neurons with Izhikevich [Izh03] neural dynamics, with random connectivity, initial states, and parameters, updating neural state once per ms. The model reproduces the results presented in [JFW08](fig. 6.1).

6.2.2 Time-Domain Performance

Further modelling uses the results from this experiment to run a cycle-accurate simulation of the top-level model (fig. 6.1) to analyse the impact of system delays on spike-timing accuracy. Figures 6.2(a) and 6.2(b) show the results. Each point in the raster plot is one spike count in the histogram. The spike-raster test verifies that there are no synchronous system side effects that systematically affect the model timing. Using the timing data from the simulation, it is possible to estimate the timing error for each of the spikes in the simulation - that is, the difference between the model-time "actual" timing of the spike and the system-time "electronic" timing of the spike - by locally increasing the timing resolution in the analytic (floating-point) Izhikevich model to 50 μs in the vicinity of a spike and recomputing the actual spike time. Most spikes (62%) have no timing error, and more than 75% are off by less than 0.5ms - the minimum error necessary for a spike to occur ± 1 update off its "true" timing. Maximum error is, as expected, 1 ms, since the update period fixes a hard upper bound to the error. In combination with the raster plot verifying no long-term drift, the tests indicate that SpiNNaker can maintain timing fidelity within a 1 ms resolution.



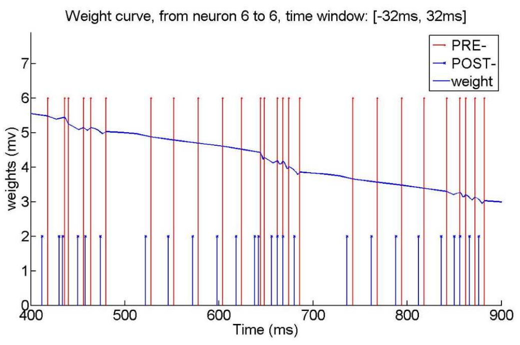
(a) SpiNNaker spiking neural simulation raster plot. The network simulated a random network of 60 neurons, each given an initial impulse at time 0. To verify timing synaptic plasticity was off.



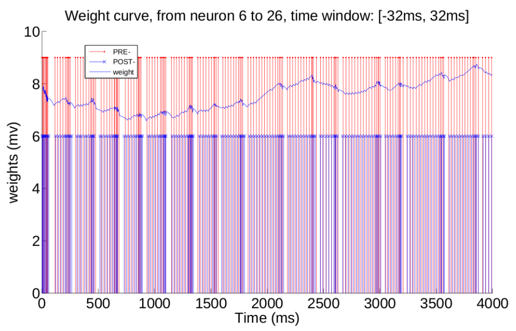
(b) SpiNNaker spike error histogram. Estimated spike-timing errors are from the same network as the raster plot.

Figure 6.2: SpiNNaker time-domain behaviour

A final series of tests [JRG+10] verified the STDP functionality by configuring a random network containing 48 excitatory Izhikevich neurons with STDP enabled, and 12 inhibitory neurons with no STDP. Connectivity was set to 83%, i.e. 40 connections per excitatory neuron. By isolating weight updates between pairs of neurons and correlating them to firing patterns, it is possible to confirm realistic behaviour. Fig. 6.3 shows the updates for two different targets of neuron 6; as expected, a self-recurrent connection weakens consistently, since presynaptic inputs always reflect the postsynaptic spike occurring an axonal delay before. Likewise, a forward connection to neuron 26 shows a variable update pattern with a long-term upward trend, consistent with the overall trend for neuron 6 to fire before neuron 26.



(a) Self-recurrent



(b) Forward

Figure 6.3: STDP updates, self-recurrent and forward connections

6.3 MLP Test Issues: Connectivity Minimisation, Load Balancing

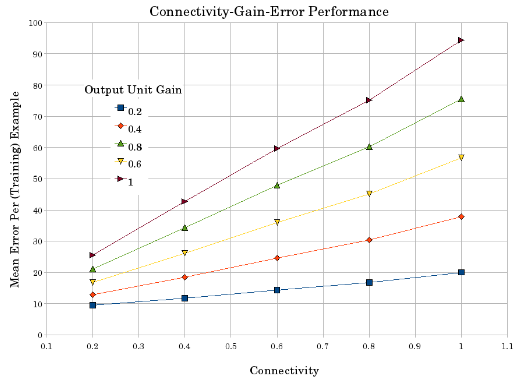
6.3.1 Model Exploration

Since the MLP network has had as a primary design goal replication of the functionality of the LENS neural network simulator, high-level model testing used LENS to run neural network simulations. The simulator ran 2 different neural networks (section 5.3.5): the LENS hand-digits example modified to include 64 input, up to 40 hidden, and 10 output neurons in a simple feedforward configuration and the larger phonetic-recognition network [WR07].

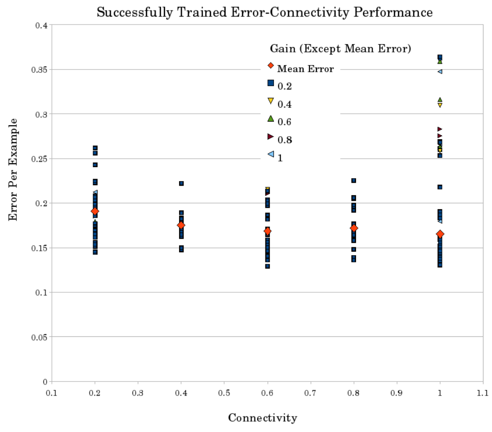
Experiments varied 3 different parameter classes: connectivity, number of neurons, and gain. To test the effect of various parameter combinations trials permuted each parameter in a series of steps. The smaller network used a parameter step size leading to 5 different values for connectivity and gain over each of the input, hidden, and output layers, a total of 15,625 possible permutations. The larger network preferentially varied the connectivities in the preference order hidden-output, hidden-hidden, output-hidden, input-hidden, input-output, and output-output over 3 different values, and hidden unit gain over 2 different values. Considering only the actual connection paths as per the architecture above this leads to $3^8 * 2^2$ - 26,244 permutations. A single trial ran each permutation for 500 training epochs in the small network, 1000 in the large one and measured the final and mean errors so as to get information both on ultimate classification performance and learning rate (using the mean as a proxy for the rate).

Results from all tests (figs. 6.4(a), 6.5(a), 6.5(b)) demonstrate an approximately linear relationship between connectivity and error performance down to a certain minimum value. For the simpler hand-digits networks performance clearly also separates according to gain. At higher connectivities, lower gains show improved performance by decreasing the slope of the error line, thus the higher the connectivity, the greater the impact of reduced gain. One possible interpretation is that reduction of gain works by minimising the error in early stages of the training. With large gains and therefore steep transfer functions small deviations in input lead to possibly large output deviations, and hence large errors if there is any error in the initial inputs.

In the large neural networks, architectural complexity makes it difficult to separate the critical parameters, although the general pattern is clear. By grouping the series first by Hidden0-Output0 connectivity, then by Input-Output0 connectivity it is possible to identify groups. Each group contains 2 branches, identifiable as a high-gain (upper) branch and a low-gain (lower) branch. Overall, the 200-neuron-per-hidden-layer networks perform better than the 50-neuron-per-hidden layer networks, suggesting the improved computing power of additional feature representations per class. One interesting feature of the 50-neuron networks is the presence of some series (i.e. parameter combination sets) whose performance at all Hidden0-Output0 connectivities is comparable to the 200-neuron networks and considerably greater than the other series. These appear to occur with certain combinations of Input-Hidden0 and Output0-Output1 connectivities, although the precise nature of the relationships has not been determined.



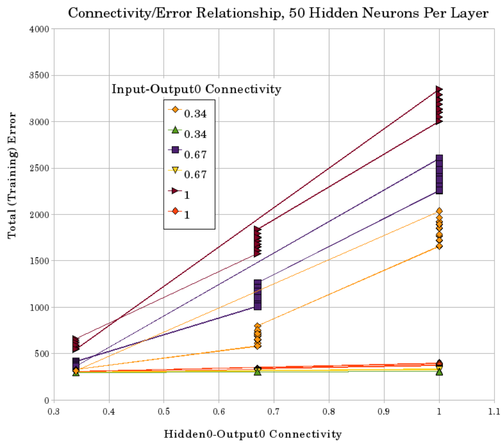
(a) Error-Connectivity results. Data series are grouped by gain.



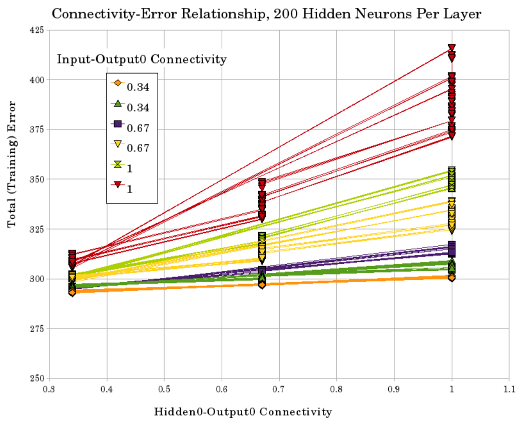
(b) Training results for configurations that successfully learned the task. Red diamonds are average values; blue squares are the individual results on each trial.

Figure 6.4: Performance of the hand-digits application.

Examination of the output performance with networks trained for the task reveals the effect of overtraining with lower connectivities. Fig. 6.4(b) gives the error with hand-digits networks that successfully learned the training set according to the maximum error criterion. These networks then ran a test set from a second, independent data source. With all connectivities the error is small, however, the mean error reaches its minimum at about 60% connectivity and increases (with greater variability) as the connectivity goes beneath this value.



(a) 50 hidden units



(b) 200 hidden units

Figure 6.5: Error-Connectivity results for the phonetic recognition network. Refer to the text for series groupings.

6.3.2 Hardware Functionality

The first test is a simple one to verify that it is possible to get the MLP network to function at all on SpiNNaker. The test creates an application based on the "digits" application from LENS. Actual network design removes extraneous structural complications from the example to arrive at a simple feedforward network with 20 input, 20 hidden, and 4 output neurons (section 5.3.5. The test initialised the weights randomly between [-0.5, 0.5]. Using the augmented training set from fig. 5.6, the network then ran through 3 successive training epochs. Results are in fig. 6.6. Weight changes show an expected evolution, compared to an identical Lens simulation on a standard PC (with floating-point values throughout). In particular, the simulation duplicates the following characteristics:

1. Weight changes on the order of 0.001.

2. A mix of increasing and decreasing weights, with an overall downward trend.

3. Regular periodic fluctuations in weight value.

Overall, these results are consistent with basic functionality.

The second series of tests establish packet-processing limits. It is important to know how far the SpiNNaker architecture can scale in terms of local packet density before breaking down. By generating additional packets from the actual traffic of the simulation proper, the MLP model can form a good test case for

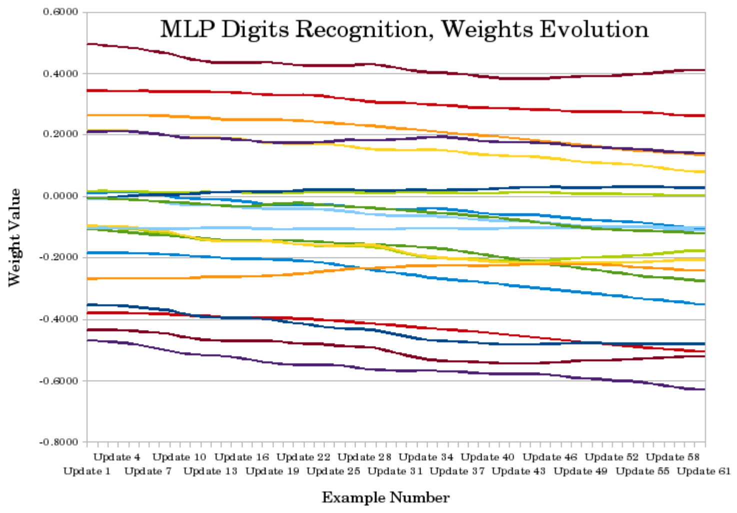


Figure 6.6: SpiNNaker MLP test, weight changes. To improve readability the diagram shows only *selected* weights; unshown weights are similar.

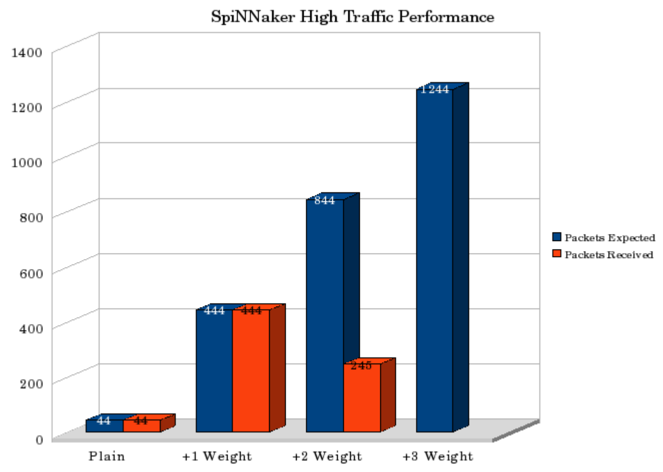


Figure 6.7: SpiNNaker packet handling performance

packet congestion. The simulation model generated a separate packet type that output the weight of each synapse, for each input. It then ramped a number of duplicates of the same packet, so that the communications controller sent a burst of n packets each time it output a weight updated, where n is the number of duplicates. Results are in fig. 6.7. SpiNNaker was able to handle $\sim$ 11 times more packet traffic without breaking down, corresponding to 1 additional packet per weight update. The network starts breaking down due to congestion by 2 packets per weight update, and by 3 packets became completely paralysed: **no** packets reached the final output. We found that the failure mode was the speed of the ISR: by 3 packets per update packets were arriving faster than the time to complete the Fast Interrupt (FIQ) ISR. Clearly, very efficient interrupt service routines, together with aggressive source-side output management, are essential under extreme loading conditions.

6.4 LIF/NMDA Issues: Model Simplicity, Temporal Synchronisation

6.4.1 Single Neuron Dynamics

Systems such as the planned attentional control network require precise synchronisation because the focus of attention is a temporally-varying object whose time and phase of salience is critical. This makes it an ideal context to test synapses with different temporal properties like fast AMPA/GABA-A versus slow NMDA synapses. In order to verify the timing characteristics of the network, the first tests characterise the timing behaviour of the (primitive) implemented LIF model.

Testing single neuron dynamics injected short pulses of current into a neuron with the following parameters: $V_0 = V_s = -75mV$, $V_r = -66mV$, $f_n = \frac{1}{4}ms^{-1}$, $R = 8$, $V_t = -56mV$. Here V_s is the reset voltage, V_r is the rest voltage, f_n is the natural frequency ($=\frac{1}{\tau}$), R is the membrane resistance, and V_t is the threshold voltage. Results in fig. 6.8 compare the accuracy of the implementation against the same neuron implemented with Brian [GB08]. The difference in the spiking region occurs because the SpiNNaker model sets $V = 30mV$ when a neuron crosses the threshold in order to generate a spike event.

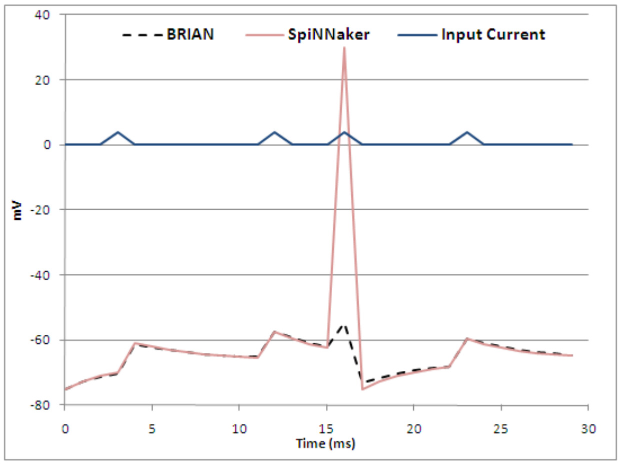


Figure 6.8: Single neuron dynamics. The test injects 4 pulses of current into the neuron. Traces compare the membrane potential of the simulation run on SpiNNaker (continuous line) with the same neuron implemented in Brian (dashed line).

6.4.2 Neural Subcircuits

The next series of tests examine the small-scale building blocks: the comparator and the oscillator. Fig. 6.9 presents the simulation results for the comparator. The network is able to discriminate the inter-pulse interval between the firings of neuron a and neuron b with millisecond precision by producing a spike from the corresponding detector neuron. This result confirms the millisecond precision of the LIF module implementation.

The oscillator network provides a good test of network dynamics. 8 neurons from the excitatory group were chosen as input neurons receiving a constant current injection of 3 nA, to make them fire approximately every 10 ms. Excitatory weights were set in order to build up the background activity of the network slowly. Once there is sufficient activity, the whole excitatory group starts firing, causing the inhibitory neurons to fire a few ms later (due to the dense connectivity). Inhibitory weights were set to quench network activity quickly. Fig. 6.10 presents the results of the simulation. In accordance with the model, the network oscillates between high activity and low activity.

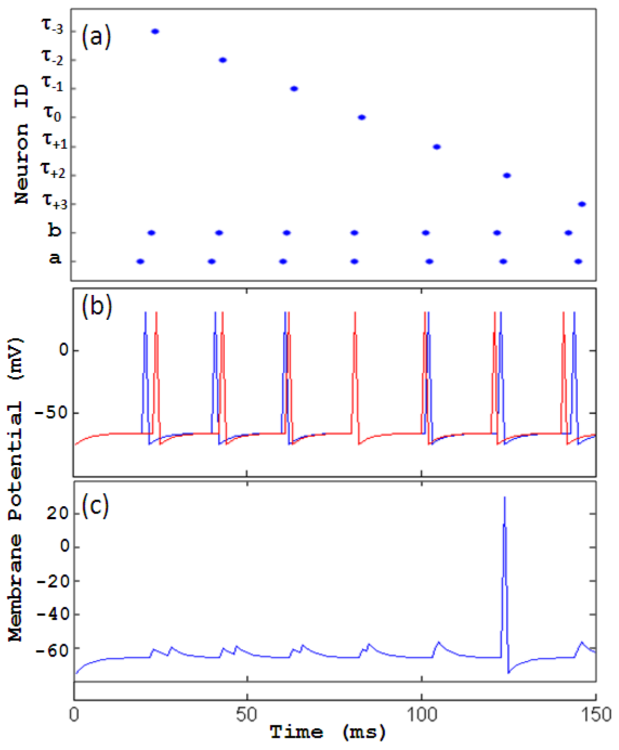


Figure 6.9: Spike Propagation. (a) Raster Plot. Only the neuron detecting the correct interpulse interval fires. (b) Membrane potential of neurons a (blue) and b (red). (c) Membrane potential of detector neuron τ_{+2}. The neuron only fires when spikes from neurons a and b converge with coherent synaptic delays

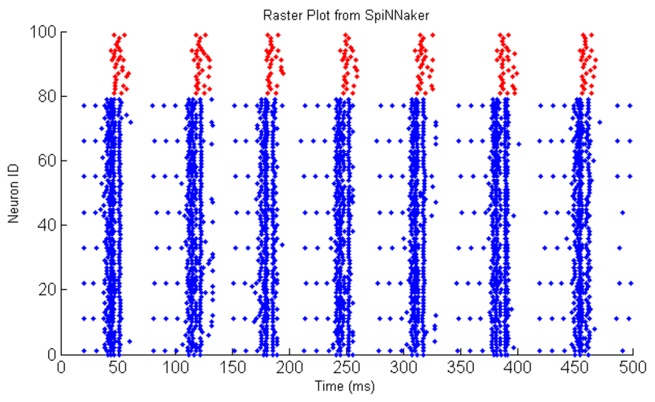


Figure 6.10: Oscillatory Network Raster Plot. Input neurons (ID's 1, 11, 22, 33, 44, 55, 66, 77) feed excitatory neurons (ID's 0-79), slowly building up the activity until excitatory neurons start to fire with high frequencies. Inhibitory neurons (ID 80-99) then activate, quenching the activity of the network.

6.5 Summary

Testing using software simulations has established the basic functionality of the 3 different neural models. It provides a fairly accurate prediction of what the hardware can and cannot do. Just as importantly, however, it makes clear what software simulation itself can and cannot do. The software model considers the simulation environment as an integral part of the entire system; a central research theme has been to place it in proper context so that potential users know what they can reasonably expect to achieve with simulation. The main findings establish the rôle of simulation:

Verification of Chip-Model Fidelity

By comparing directly against high-level simulations in MatLab (for the Izhikevich model) Lens (for the MLP model) and Brian (for the LIF model) it has been possible to show correspondence between the on-chip models and their abstract mathematical representations, at least within the 16-bit fixed-point representational precision adopted in SpiNNaker.

Estimation of Hardware Performance

It is possible using SoC Designer and SystemC to run simulations of small-scale networks: these can reveal potential performance issues although they cannot answer questions of scalability and network traffic conclusively. The MLP model tests are a particularly good example of this use.

Analysis of the Effects of Parameter Variation

Software improves data visibility: a critical concern for studies of the effect of different parameter settings such as the connectivity examination. Interestingly, reducing connectivity to the minimum levels that support the necessary functionality not only improves network traffic performance but appears to improve learning times, at least with MLP networks - this result helps to justify the connection-limited architecture of SpiNNaker.

Evaluation of the Limits of Simulation

Where simulations need long run-time in order to verify accuracy completely, software is an inefficient approach: real-time simulations of minutes or hundreds of epochs may take days to complete under a SystemC model. Overall, then, software simulation is very useful for pre-implementation verification, using simplified networks that capture the essential aspects of a candidate model, but is impractical for simulation even of small parts of full-scale working networks that do anything beyond synthetic “toy” applications.

Validation of On-Chip Model Implementations

Overall, the conclusion is that the ideal rôle of software simulation is conceptual model testing: finding out whether a given neural model will work at all (which is the approach that has been adopted here), as opposed to model execution: running “live” simulations. It remains to test models on physical SpiNNaker hardware in order to examine performance and dynamics in real-world applications.

Chapter 7

Phase III: Execution - Hardware Testing

7.1 Plan

Hardware testing has the obvious goal of verifying the physical operation of the hardware, and in the case of a general model library performs the equally critical task of evaluating model flexibility and scalability. This means, in essence, how quickly can the system switch between different neural models, of what size? The ideal is heterogeneous operation: the ability to run different models, of significant size, on different processors within the same system. Speed is also an important consideration; this takes the form of real-time fidelity in the case of dynamic spiking networks, while in the MLP and other similar networks the task is simply to establish how many updates the system can sustain per second. Finally, because hardware testing can run large-scale models for long enough periods of time to demonstrate behavioural dynamics, it is appropriate to compare the hardware results against abstract software models. These tests will reveal both the behavioural correspondence and any quantitative differences, so that it is possible to explain (and possibly adjust for) hardware output with reference to an "ideal" abstract model.

This chapter runs the basic models introduced earlier on the SpiNNaker hardware platform. It will demonstrate the ability of the configuration system to specify and run the different models with hardware acceleration, for networks of reasonable size (500 neurons in the case of spiking networks). All implementations start with a high-level PyNN description and follow the library-based automation process to instantiate the models on-chip. The most important evaluation priority is size and heterogeneous model switchability, and the tests will emphasize this. Where possible they also attempt to compare model output directly against reference simulators. All tests ran on the SpiNNaker test platform, and are thus somewhat preliminary, but even on this reduced system the speed at given size exhibits substantial improvement over software models, while perhaps even more significantly, the system could be reconfigured in seconds, making the SpiNNaker platform equivalent to a software system with hardware-scale acceleration and scalability.

7.2 The Izhikevich Model in Hardware

7.2.1 Comparison with Brian

To test the Izhikevich neuron model the system ran a 3 Layer feedforward network containing 21 Izhikevich neurons, implementing the Evaluator's association layer (section 5.2.3). Layer 1 has one (input) neuron which receives constant DC current injection consistent with a 20mV membrane potential, making it fire at approximately 40Hz. Layer 2 and Layer 3 consist of two populations of 10 Neurons each. Layer 1 connects to all neurons in Layer 2 with random delays. Layer 2 connects to Layer 3 with one-to-one connections having random weights.

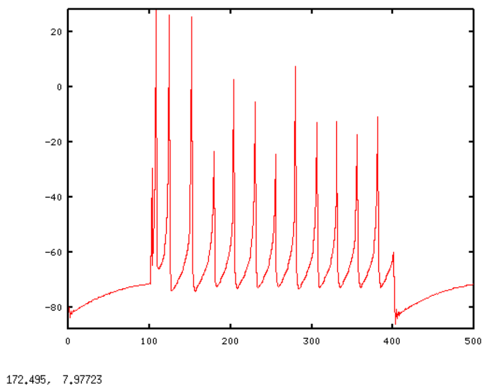
A Brian script (Andre van Schaik, personal correspondence, script included) tested single neuron dynamics. SpiNNaker results qualitatively preserve the Brian dynamics, confirming the validity of the fixed point implementation of the model on the SpiNNaker in the presence of a high firing frequency. Results are as follows (figs. 7.1(a), 7.1(b)):

Examining the membrane potential for a neuron in Layer 2 and two neurons in Layer 3 makes it possible to verify correct output propagation and setup of the random weights and delays. Spikes arriving from the input neuron in Layer 1 to a neuron in Layer 2 cause it to fire after receiving approximately 4 spikes (fig. 7.2(a)). Spikes arrive at two different neurons in Layer 3 with different times and strengths, according to the parameters for each (single) synaptic connection (fig. 7.2(b)).

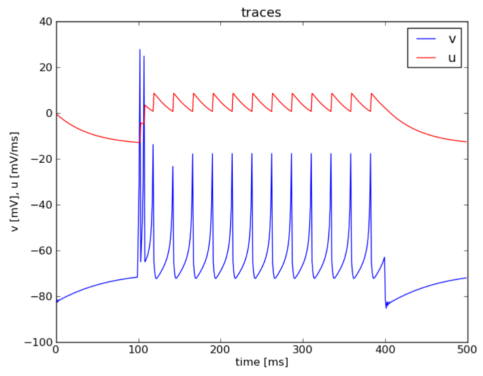
7.2.2 Real time simulation of a 500 neuron network

To simulate a larger network, the system ran the network containing 500 Izhikevich neurons from section 5.2.3. While testing the ability of the SpiNNaker system to run a large simulation in real time, it also verifies its capability simultaneously to store and report the necessary data to generate a raster plot at the end of the simulation. The network aims to scale down and reproduce the results of the first part of the simulation in [IH09], exhibiting delta-frequency oscillations of ~2-4 Hz. The model ran for 1 second real time, producing the following raster plot at the end of the simulation (fig. 7.3).

```
  '''
Izhikevich Neuron Model

Simple Model of Spiking Neurons
Eugene M. Izhikevich
IEEE Transactions on Neural Networks, 2003
'''
from brian import *

defaultclock.dt = 0.01*ms

# parameters
a = 0.02/ms
b = 0.2/ms
c = -55*mV
d = 2*mV/ms
C = 1*nF
stim = 5*nA
N = 1

eqs = Equations('''
dv/dt = (0.04/ms/mV)*v**2+(5/ms)*v+140*mV/ms-u+I/C    : volt
du/dt = a*(b*v-u)                                      : volt/second
I                                                      : amp
''')
reset = '''
v = c
u += d
'''
threshold = 30*mV

G = NeuronGroup(N,eqs,threshold=threshold,reset=reset)

G.v = -70*mV
G.u = b*G.v
v = StateMonitor(G,'v',record=0)
u = StateMonitor(G,'u',record=0)

G.I = 0*nA
run(20*ms)
G.I = stim
run(400*ms)
G.I = 0*nA
run(20*ms)

figure(1)
title('traces')
plot(v.times/ms,v[0]/mV,'b',u.times/ms,u[0]*ms/mV,'r')
xlabel('time [ms]')
ylabel('v [mV], u [mV/ms]')
legend(('v','u'),loc='upper right')
figure(2)
title('phase diagram')
plot(v[0]/mV,u[0]*ms/mV,'b')
V = linspace(-80,40,num=120)
plot(V,0.04*V**2+5*V+140+stim/nA,'r')
plot(V,0.04*V**2+5*V+140,'k')
plot(V,b*ms*V,'c')
axvline(30,0,1,color='g')
xlabel('v [mV]')
ylabel('u [mV/ms]')
legend(('trajectory','max stim nc','no stim nc','u nc',
'threshold'),loc='upper right')
axis([-80,40,-20,0])
show()
```

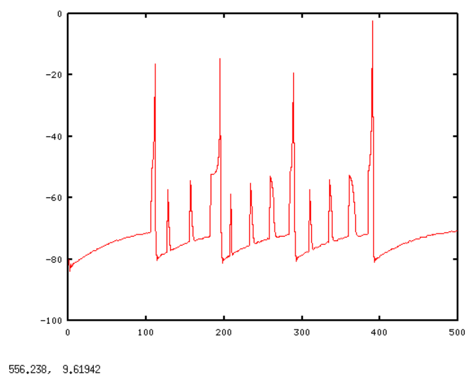


(a) SpiNNaker output

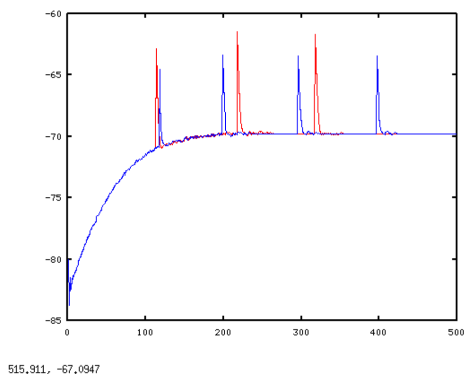


(b) Brian simulator output

Figure 7.1: Izhikevich network: Brian script and associated membrane potential output for an input neuron



(a) A neuron in Layer 2



(b) Two neurons in Layer 3

Figure 7.2: Izhikevich model: Membrane potentials with connections having random delays and weights.

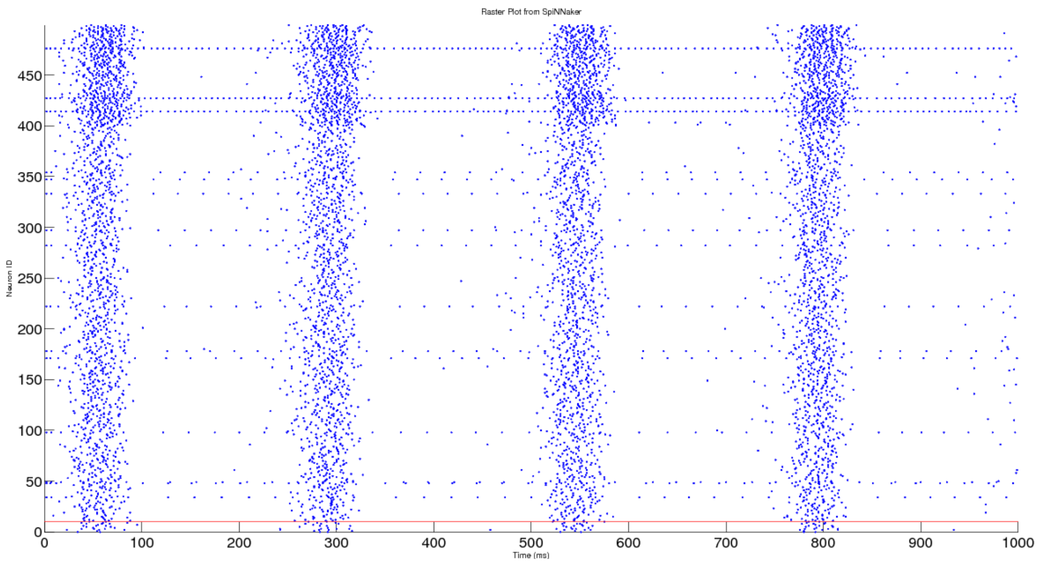


Figure 7.3: Raster plot for the 500 neuron Izhikevich model simulation

7.3 The LIF Model in Hardware

7.3.1 Single neuron testing

To validate the SpiNNaker LIF implementation at a low level in hardware, an initial test simulated a single neuron in three environments: the SpiNNaker chip, a Brian script, and a NEST [PEM+07] simulation. Stimulating it with the same current shows a close match between the neuron dynamics for SpiNNaker, Brian, and NEST. (fig. 7.4). This test was able to run immediately after the Izhikevich single-neuron test, using a single script-driven configuration command to reconfigure the chip and load the new model within 6 seconds.

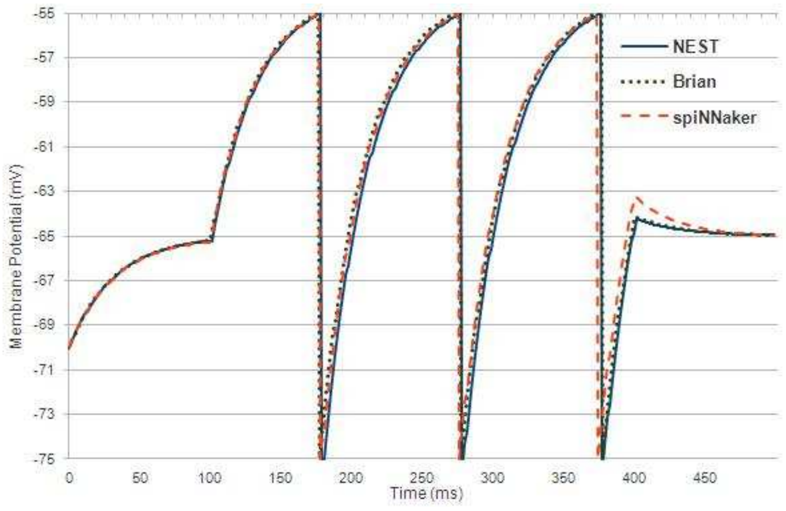
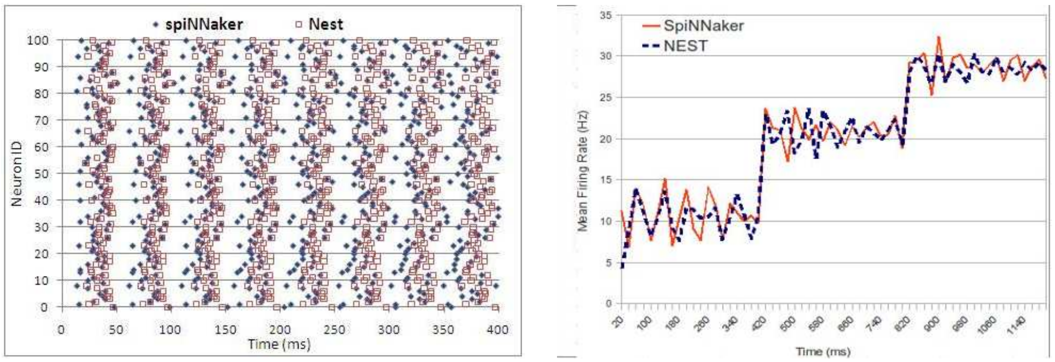


Figure 7.4: SpiNNaker output membrane potential for an LIF neuron in the input population

7.3.2 Network tests

To test network dynamics at a larger scale, a larger network simulated 500 LIF neurons. 400 neurons were excitatory, 100 inhibitory, and the neurons all had the following parameters: $V_i = -60mV, V_r = -49mV, V_s = -60mV, V_t = -50mV, \tau = 32ms$. The population had random connectivity with 2% probability of connection. A subset of 40 neurons received a current input injection; this current increased in 2 step-jumps at 400 and 800 ms. Figure 7.5(a) shows a raster plot of selected neurons in the population (for clarity), comparing the results from SpiNNaker versus corresponding results from NEST. Qualitatively, the neurons show pulsed waves of activity of similar frequency and phase. Results from NEST show somewhat more coherent behaviour; this is an expected result of the differences in synaptic dynamics. SpiNNaker synapses use "all-or-nothing" activation with instantaneous current injection, in contrast to NEST which uses current-based synapses with a linear first-order kinetic, i.e. an exponential decay of activation. The effect of SpiNNaker's instantaneous kinetic is to advance the spike timing on a per-neuron basis, since the synapse injects its entire current immediately, thus increasing the membrane potential maximally at the time of injection. Effectively the current-based synapse performs a form of small-time-constant low-pass filtering, smoothing out local variations in neuron output timing.

An examination of the population-based effects reveals the quantitative behaviour more clearly. Figure 7.5(b) shows a comparison of the mean firing rate of the population for both the SpiNNaker and NEST simulations. The graph clearly reveals the presence of synchronous waves through the oscillations in mean firing rate. Despite observed qualitative differences, quantitative match in mean firing rates between the two models remains good, peaks and troughs matching in frequency and phase except for one deviation between 500 and 600 ms where the SpiNNaker system lags in phase. The apparent interpretation is that a more coherent spike input in the NEST system leads to an earlier group peak, since neurons receive spike inputs in a more tightly confined time window.



(a) Raster plot of selected neurons (b) Mean firing rate

Figure 7.5: 500-neuron LIF population: Comparison of SpiNNaker versus NEST

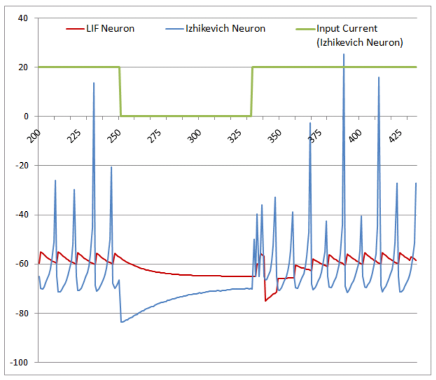
As a final, behavioural verification of the model, students from the SpiNNaker group implemented a network to control a robot in a line-following task, using spiking input from an AER-based silicon retina. The network contained 532 LIF neurons in 3 groups. This network was successfully able to control the robot, enabling it to follow a line reliably over a distance of ~1 m. In addition to verifying the on-chip model implementation in a reasonable real-world task, these tests also demonstrate the larger system benefit of the event-driven model using an AER interface: it was possible to integrate different devices, from different research groups, to create a functional system based purely on the neural model of computation.

7.4 Heterogeneous On-Chip Modelling

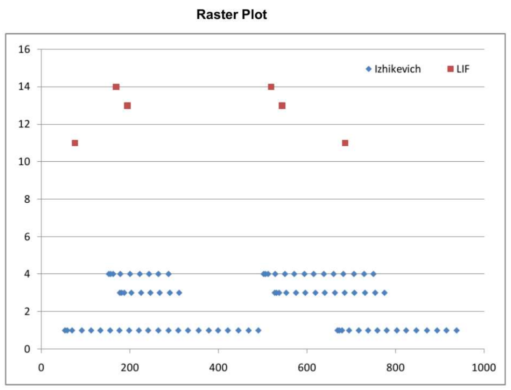
7.4.1 Mixed LIF/Izhikevich model networks

Two additional models verified the core abstractional ability of the library: the ability to generate and run models containing heterogeneous mixes of neural types. The first simulated network used two populations, containing respectively LIF neurons and Izhikevich neurons. The network contains two layers of 15 neurons each. The first layer serves as an input layer using Izhikevich Intrinsic Bursting neurons. The second layer is an LIF layer, configured so that neurons will emit spikes only if stimulated over a certain frequency. A modulated stimulus injected a current of 20 into each input neuron. Each Izhikevich neuron emits a burst of 3 spikes, then spikes regularly (fig. 7.6(a)). In turn each LIF neuron thus spikes only when it receives the initial burst from the lower level population (fig. 7.6(b)).

The second network implemented the output-reinforcement layer for the Evaluator, configuring 3 layers of 15 LIF, Izhikevich and LIF neurons respectively. The Izhikevich neurons are Chattering types that generate a burst "clock" to gate input from the Layer 1 LIF neurons, so that the Layer 3 LIF neurons will only fire when an input from Layer 1 is coincident with the burst window (fig. 7.7). Burst-type neurons are a useful and biologically plausible way of supplying reinforcement: by creating an input "window" in downstream neurons that gates other input sources. This network demonstrates the important ability to generate control signal-like inputs to integrating layers.



(a) Membrane dynamics



(b) Raster plot

Figure 7.6: Membrane and network dynamics of the first test network. Parameters were: for the LIF neurons $\tau_m = 32ms, V_r = -65mV, V_s = -75mV, V_t = -55mV$; for the Izhikevich neurons, $a = 0.02, b = 0.2, c = -65mV, d = 6$.

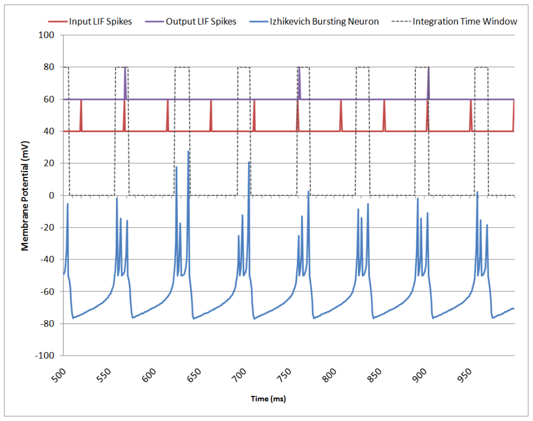


Figure 7.7: Second test network dynamics. Parameters were: for the LIF input neurons $\tau_m = 16ms, V_r = -49mV, V_s = -70mV, V_t = -50mV$; for the Izhikevich neurons, $a = 0.01, b = 0.2, c = -50mV, d = 5$; for the LIF output neurons $\tau_m = 16ms, V_r = -65mV, V_s = -75mV, V_t = -55mV$.

7.4.2 Hardware scalability

From the models that have successfully run it is clear that SpiNNaker can support multiple, very different neural networks; how general this capability is remains an important question, particularly concerning the scalability of the low-level neural library primitives. The LIF model's core computation time of 10 instructions without spiking, 21 with, is probably the minimum possible for any model with broad acceptance within the spiking neural model community. Implementing LIF and Izhikevich models along with two different synaptic models (STDP and NMDA) has made it possible to examine efficiency limits within the SpiNNaker hardware. The investigation considers a synthetic network containing an identical number of neurons per processor. Neurons randomly connect to other neurons in the same population with random initial weights and delays. The total number of connections is set so that each neuron receives the same number of inputs. Where the number of inputs from other (internal) neurons is less than the specified number, the neurons receive additional (external) inputs sufficient to make up the balance of inputs. A randomly-selected 1% of neurons and external inputs receive a synthetic stimulus: a regular, fixed-frequency spike train. Each network is configured with a single fixed neural and synaptic type, uniform throughout the population. The investigation examines 5 combinations of neural/synaptic type, 3 levels of connectivity, and 3 input frequencies. Respectively, these were LIF/fixed synapse, LIF/STDP, LIF/NMDA, Izhikevich/fixed, and Izhikevich/STDP; 100, 1000, and 10,000 inputs/neuron; and 1, 10, and 100Hz. By varying the total number of neurons it is possible to examine the maximum number of neurons a single SpiNNaker processor could model (fig. 7.8).

This examination considers the model to "fail" if one of the following conditions apply: the amount of available memory is insufficient to support the number of neurons/synapses; processing of neural state is still incomplete by 0.2 ms before the next (1 ms) Timer interval; the processor halts (because of packets arriving faster than the interrupt routine can service them). For most models, memory capacity rather than processing overhead is the limiting factor: most configurations easily supported rates in excess of 2000 neurons/processor·ms; considerably over the memory limit of 1724 AMPA/GABA-A-only neurons per processor, or 910 neurons per processor with NMDA synapses. Of the various factors the mean spiking rate of the active population has the greatest impact on performance.

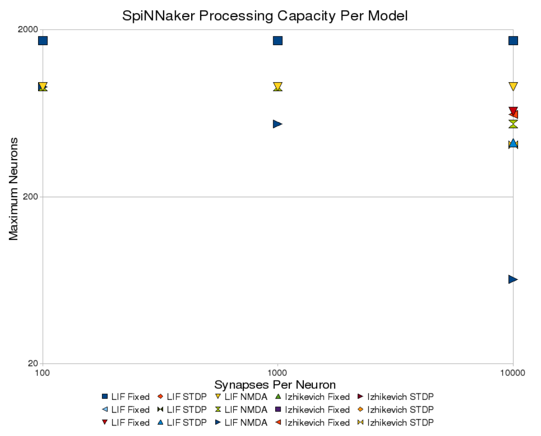


Figure 7.8: Number of Neurons Per SpiNNaker Processor

This was particularly noticeable for NMDA synapses where high firing rates increase the number of iterations necessary to precompute future NMDA activation; a multiplicative effect. High rates also increase the mean activation, making it more probable that the NMDA gate will turn on, increasing the downstream firing rate. The result is a processing cascade - and at the maximum input rate of 100 Hz the number of neurons a processor can model drops drastically. Careful analysis of the flow of execution on the SystemC model determined that the failure mode was the speed of the ISR: packets were arriving faster than the time to complete the Fast Interrupt (FIQ) ISR. In the hardware simulations, a similar analysis showed that network breakdown in the spiking models was happening due to receive buffer overflow. Some of this may be attributable to known inefficiencies in the packet queue implementation that is an important part of the deferred-event model. It is very clear, therefore, that careful pre-implementation modelling of the population dynamics is essential, and complex, biologically realistic synaptic models in particular need aggressive optimisation. A neural component library containing precompiled, pre-optimised models becomes particularly valuable in this context.

7.5 Summary

Successful implementation of the networks on-chip demonstrates the functionality of the design-automation tools: the ability, starting from a high-level model written in PyNN, to use scripted automation based on library components to transform the description into a SpiNNaker instantiation. Furthermore, these tools permit direct comparison against the reference high-level models written in the Brian and NEST software simulators. This "closes the loop" in the network modelling cycle, fulfilling the primary objective of the event-driven library architecture: an *integrated* network modelling and simulation system for dedicated neural hardware. The main results are:

Scalability

Using the SpiNNaker test chip in a four-chip configuration, it has been possible to test networks of reasonable size: up to a 4500-neuron system. These models ran in real-time, and in the case of the robot, in a real-world environment. However, there are scalability concerns with complex models containing synaptic dynamics; when spike rates are high the resultant traffic can overwhelm the receiving processors' ability to keep up.

Heterogeneous Model Support

The Izhikevich and LIF models ran successively on the same system, within the same testing session. Switching the model took less than 30 s, using the automated tool chain. It was also possible to configure the system to run both models concurrently, within the same system, from within the automated, library-driven development system.

Fidelity

Single-neuron model fidelity versus reference models is nearly a match fit and the large-scale dynamics have similar properties. Where large-scale dynamics differ in detail, e.g. in the 500-neuron LIF model, results are consistent with predictions based on the differences between the reference model and the on-chip model. The ability to use the models within a behaving environment: a robot in a line-following task, indicates that quantitative differences are probably not significant for behaviour, at least not at the gross level.

Chapter 8

Implications: Spatiotemporal Abstraction, Complexity Limitations

8.1 Agenda

A large-scale system such as SpiNNaker raises almost as many questions as it can answer; this begs the question as to what its purpose really is. This chapter attempts to provide some answers. Undoubtedly the primary purpose is **abstraction of the neural model of computation**. This has been a recurring theme throughout this work. The first section discusses what new abstraction capabilities SpiNNaker provides, with reference to concrete examples that show how implementing different models on a physical device rather than a software formalism reveals new insights. In large part this is because the SpiNNaker platform moves away from the synchronous, sequential model that underlies conventional simulation and introduces processing assumptions that may be inappropriate for neural computing. Parallel, event-driven processing overturns any assumptions about control-flow order, data determinacy, or global coherence: these are the questions for neural modelling that systems like SpiNNaker are ideal for exploring.

Another obvious purpose is simply model scaling. A large, hardware platform such as SpiNNaker makes possible much larger neural models than software has been able to simulate, but the question still remains about how big the models can go. The second section explores this question. Thus far simulations in software and hardware have been able to indicate where the challenges lie, but clearly many of the questions will remain unanswered until the full-scale system begins to simulate very large networks. Managing the communications traffic emerges as the major challenge. Some of this may be possible with clever algorithmic and mapping techniques, but undoubtedly much of it will rely on intelligent choices in model abstraction. Such strategies move from an optimisation to a necessity when the scale approaches that of the brain: the third section looks at that scaling roadmap. One of the other recurring themes of this work has been tool automation: for scaling to be successful when neural networks reach billions of neurons, however, it will need an entirely different order of tool automation. Fully statistical methods that can take advantage of the SpiNNaker hardware not only to run the simulation but indeed to generate the model will be a central component of these tools. A similar transformation of user data visualisation and analysis is likewise inevitable. Although such tools and models lie beyond the scope of this work, they represent the fruition of its aim: computational cognitive neuroscience: the ability to deduce the function and structure of the brain, from abstract computational models that reproduce its behaviour.

8.2 Abstract-Space, Abstract-Time Models

The central theme of the SpiNNaker system is *process abstraction*: the ability to decouple the neural model under simulation from the implementation details of the hardware. The tests, of 3 different models with different tasks, on the same platform and using the same development environment illustrate how to construct a standardisable platform that makes it possible, at least in principle, to model any neural network in hardware. This universal capability is beneficial, indeed essential, for experimenting with different models at large scale, but it points to a more important development: abstract neural computation. If one of the primary goals in neural modelling is to understand the model of computation, it must be possible to develop abstractions of neural function or the entire process is reduced to phenomenological observation. Thus to verify any model of neural computation it is necessary to implement it in a reproducible form that does not depend on low-level details. *The* central research problem of neural networks is to develop a universal abstract model of computation that retains the essential characteristics of neural networks while eliminating unnecessary individual features.

In this regard, the most important architectural principle of the software model, characteristic of native parallel computation, is **modularisation of dependencies**. This includes not only *data* dependencies (arguably, the usual interpretation of the term), but also temporal and abstractional ones. The model does not place restrictions on execution order between modules, or on functional support between different levels of software and hardware abstraction. Architecturally, the 3 levels of software abstraction distribute the design considerations between different classes of service and allow a service in one level to ignore the requirements of another, so that, for example, a Model level neuron can have a behavioural description that does not need to consider how or even if a System level service implements it. From a process-flow standpoint, it means that services operate independently and ignore what may be happening in other services, which from their point of view happen “in another universe” and only communicate via events “dropping from the sky”, so to speak. Such a model accurately reflects the true nature of parallel computing and stands in contrast to conventional parallel systems that require coherence checking or coordination between processes.

Up to now it has only been possible to do this type of abstract neural modelling in software: hardware has been process-specific and structured at a low level. In essence, neural hardware systems have presumed to understand, or assumed, what the neural model of computation is *a priori*. Software, however, has had the problem that as the network size scales, either the level of abstraction must increase, eventually to the point where it starts to lose essential features of the neural model of computation; or the simulation must slow down, eventually to the point where it is simply too slow to run in practical time. With the SpiNNaker development system it is possible to create and test large-scale, abstract neural models as candidates for the "formal" universal model of neural computation, without sacrificing behavioural detail where necessary. This system gives the modeller the tool to access and observe the actual principles of neural computing. How general this capability is, however, remains an important question.

8.2.1 Abstract Representation

Implementing a function pipeline provides a standard "template" for library components. The model emerges from a consideration of what hardware can usually implement efficiently in combination with observations about the nature of neural models. Broadly, most neural models, at the level of the atomic processing operation, fall into 2 major classes, "sum-and-threshold" types, that accumulate contributions from parallel inputs and pass the result through a nonlinearity, and "dynamic" types, that use differential state equations to update internal variables. The former have the general form $S_j = T(\Sigma_i w_{ij} S_i)$ where S_j is the output of the individual process, T is some nonlinear function, i are the input indices, w_{ij} the scaling factors (usually, synaptic weights) for each input, and S_i the inputs. The latter are systems with the general form $\frac{dX}{dt} = E(X) + F(Y) + G(P)$ where E, F, and G are arbitrary functions, X is a given process variable, Y the other variables, and P various (constant) parameters. At an abstract level, the pipeline breaks these equations into a series of independent, atomic operations that can be driven by input events.

At a concrete level, the pipeline considers what functions SpiNNaker can implement efficiently. SpiNNaker's processors can easily compute polynomial functions but it is usually easier to implement other types, e.g. exponentials, as a look-up table with polynomial interpolation. Such a pipeline would already be sufficient for sum-and-threshold networks, which self-evidently are a (possibly non-polynomial) function upon a polynomial. It also adequately covers the right-hand-side of differential equations. For very simple cases it may be possible to solve such equations analytically, but for the general case, Euler-method evaluation appears to be adequate. Creating a new component for the library is simply a matter of plugging in appropriate models for the pipeline stages, allowing for extensive software reuse because most of the support libraries, low-level utilities, and "housekeeping" code can be general across all models. Only the dynamics need change. The library therefore takes the form of a general infrastructure with model-specific core routines.

In principle, then, SpiNNaker can implement virtually any network. In practice, as the packet experiments show, traffic density sets upper limits on model size and speed. Furthermore, processing complexity has a large impact on achievable performance: more complex event processing slows the event rate at which SpiNNaker can respond. At some point it will drop below real-time update. Careful management of memory variables is also an important consideration. The models involve multiple associative memories and lookup tables. If speed is critical, these must reside in DTCM or ITCM, placing a very high premium on efficient table implementations. If it is possible to compute actual values from a smaller fixed memory block, this will often be a better implementation than a LUT per neuron.

Data representation and encoding is the other challenge. Given that efficient internal representations on SpiNNaker are nonintuitive and prone to be confusing, it is essential to the *user* that the tools hide all the scaling behind automatic translation routines that scale variables from and to units the user expects. For example, in the NMDA model, the transformation $Mg^{2+} \rightarrow 16ln\frac{Mg^{2+}}{3.57}$ is far from obvious, but improves performance dramatically since it can be precomputed. It is natural to do this precomputation in the instantiation scripts, not only hiding it from the user but also reducing the risk of errors in manual parameter translation. From a *developer* point of view, however, tracking the changes in scaling presents one of the most significant design decisions. For example, in the MLP model, what is the appropriate scaling for the weight units? The implementation developed a rule that permits the developer to make reasonable scaling choices, however, with 16-bit precision as compared to the 64-bit floating-point precision typical of software-only MLP networks, there may still be a considerable amount of model-dependent verification to perform. To a large extent, confirming that signals do not overflow frequently can only be done experimentally. It is possible to envision a "characterisation" stage where the developer compares SpiNNaker results against a variety of models with different parameters. The hierarchy of simulators with different levels of abstraction becomes a significant advantage in this scenario.

8.2.2 Abstract Space

Spatial abstraction is one of the most obvious characteristics of neural modelling; indeed, in some areas the term "connectionist computing" is synonymous with neural networks. Nonetheless, they do not behave as "hard-wired" circuits, for the most part, and there is no evidence that different brains have the same precise circuit components down to the level of individual connections. There is, however, considerable evidence that the brain has characteristic patterns of connectivity: cortical columns, hippocampal place cells, visual receptive fields. Presumably this structured connectivity is achieving something useful. SpiNNaker's virtual topology provides a useful concrete context in which to explore the actual topologies of neural networks. Its topological flexibility is large but not arbitrary, and it is instructive to consider how its limitations can offer insights into the biological problem of connectivity patterns.

The NoC's finite communications bandwidth lends strong motivation to examining the effects of partial connectivity upon network performance. Experimental results with the MLP model show that partial connectivity can achieve faster training times, even in neural networks of relatively small size. There are several possible contributors to this effect. One possibility is symmetry breaking. A large network with full connectivity is initially class-undifferentiated and must spend a certain time during early learning to develop weights that bias individual neurons towards identification of a particular class or feature. The usual solution to this problem is to randomise the weight values. However, particularly with large networks, randomised weights at best give a weak asymmetry since a neuron receives inputs from many synapses. Experimental results confirm that the symmetry breaking effect is relatively weak in networks of the size examined. If the effect were strong, one would expect reduced connectivity in *any* interlayer connection to produce improved error performance, when in fact the improvement is most dramatic and obvious in the hidden-output connections. Such a result is

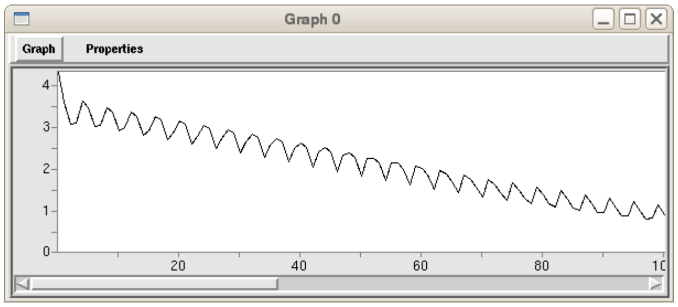


Figure 8.1: Error from the digits recognition task during early training. The oscillatory nature of the response in this fully-connected network is clear.

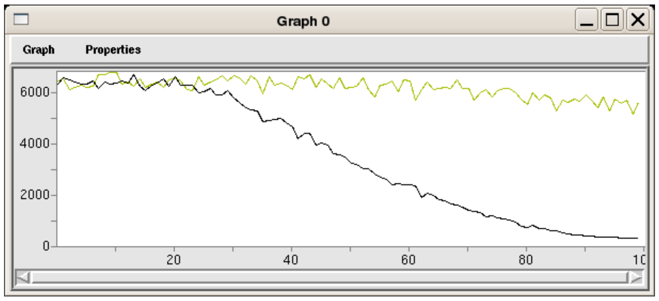


Figure 8.2: Settling time during early training. X-axis is the weight update number, in 10's (thus the tick mark "20" is the 200th update) Y-axis is the global network error. The light-green trace is the fully connected network. The black trace is the best of the sparsely connected networks

more consistent with the second possibility: minimisation of error propagation.

The theoretical model for connectivity performance predicts that reduction in the backpropagation of errors in delta-rule learning will reduce training time and produce lower error per example. In a fully-connected network, if certain neurons in the output have large errors they can propagate throughout the network, resulting in large weight corrections in potentially irrelevant connections. Particularly early in training, the large error will tend to dominate the sum in neurons in the previous layer, propagating back through the network as a wave of large weight changes. Thus in a highly connected network, the overall error initially will fluctuate before the weights settle enough that inputs from irrelevant neurons have very small values. It is easy to observe this effect even in small networks like the digits-recognition network (fig. 8.1). Training with the

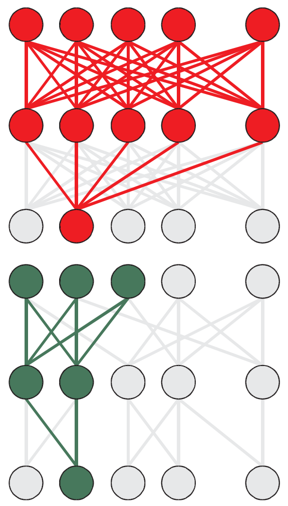


Figure 8.3: Error propagation in fully- versus partially-connected networks. In the full case, the error propagates throughout the network from even a single output with inaccurate value. Errors are propagated under partial connectivity, but remain confined to specific paths where there exist connections, leaving large sections of the network unaffected.

larger phonetic-recognition network also demonstrated this characteristic. High-connectivity networks would tend to spend several epochs with oscillating or at best marginally decreasing error before starting to learn (fig. 8.2). By contrast a network with partial connectivity confines any weight adjustments to the neurons in the path, and since there are fewer, each one has a greater probability of contributing meaningfully to the total error. The effect is two-fold: first, it helps to suppress overadjustment of weights not involved in the error - "innocent bystanders" - as it were, and second it more rapidly adjusts the erroneous inputs towards a correct value, so that training is more effective in the early epochs (fig. 8.2). Both in the forward and backward passes it is reasonable to infer that the effect of partial connectivity is to localise signal propagation into class-specific groupings (fig. 8.3).

The optimal connectivity tends to be lower with smaller network size, and beyond a critical point the performance of networks with very sparse connectivity drops dramatically. These results have a simple interpretation: at some point the number of connections and/or neurons is too small to represent the data accurately. As the model removes more connections, the network must eliminate classes or features. If, rather, the number of *connections* is fixed, and the number of neurons (hence the mean connectivity per neuron) varies, it might be equally possible to represent the same data to the same degree of accuracy with 2 networks having a different number of neurons but identical number of connections. Comparison of best-case performance for the 50-hidden and 200-hidden neuron large phonetic recognition networks supports this hypothesis. However, the smaller network will have a significant disadvantage: diminished fault tolerance and ability to represent new classes. In the limit, the smallest possible network will use every neuron and every synapse completely to represent the training data, thus if a single neuron or synapse fails there is some data loss. By contrast, if the number of neurons in hidden layers is large, the loss of one does not necessarily imply permanent and total loss of a represented class or feature. One benefit of large networks therefore lies in robustness under component failure and ability to adapt to new, heretofore unknown inputs. It is clear from the data that the larger network performs better under a wide variety of parameter combinations and hence is the preferred choice in a system such as SpiNNaker where processing is cheap and connections expensive. When examining connectivity, it is therefore preferable to treat the problem as connection-constrained rather than neuron-constrained. This result might not have been thought of, much less discovered, without the need to translate a network description onto a specific hardware platform.

8.2.3 Abstract Time

The ability to create high-level spatial representations of neural networks is powerful but already existed (in small-scale form) with FPGA neural networks; not so obvious or inevitable is the ability to abstract time. It is only recently, indeed, that research has begun seriously to attack the question of the relevance of the time domain to neural networks and it is now clear that it is not reasonable to take the model of time as a given. The event-driven model of the SpiNNaker library represents a break from clock-driven synchronous neural networks models. In synchronous systems, the fixed clock sets an absolute model of time that lies at such a low level of the model that it is effectively invisible to and often overlooked by the modeller. Therefore, it is at least possible that a simulation using a conventional synchronous sequential computer may *never* be able to model biological neural networks with full realism. Pairing the event-driven software model with SpiNNaker's asynchronous hardware creates a platform for temporal as well as spatial abstraction.

An important observation of the tests is that in a system with asynchronous components, the behaviour is nondeterministic. Thus in spiking simulations, while most of the spikes occurred without nominal timing error, some had sufficient error to occur at the next update interval(fig. 6.2(b)). The effect of this is to create a ± 1 ms timing jitter during simulation. Biological neural networks also exhibit some jitter, and there is evidence to suggest that this random phase error may be computationally significant [TS01]. It is not clear whether asynchronous communications replicates the phase noise statistics of biological networks, but its inherent property of adding some phase noise may make it a more useful platform for exploration of these effects than purely deterministic systems to which it is necessary to add an artificial noise source. In addition, it is possible to (statically) tune the amount of noise, to some degree, by programming the value of the update interval (that acts as an upper bound on the phase noise). A GALS system like SpiNNaker therefore appears potentially capable of reproducing a wider range of neural behaviours than traditional synchronous systems, while supporting the programmability that has been a limiting factor in analogue neuromorphic devices.

The deferred-event model has proven to be the most powerful event-driven mechanism for temporal abstraction . Even with nondeterministic signal timing, the deferred-event model permits not only real-time signal timing resolution but also more efficient processor utilisation, since each processor can schedule its updates according to the model-time rather than the hardware-time event sequence. It is the only successful model to date that can process event-driven neural and synaptic dynamics dependent upon contingent future events without resorting to approximate methods. The obvious application is the one for which the model was originally designed: achieving true real-time performance by distributing the processing in time. Less obvious is the ability to run multiple independent time domains concurrently. The NMDA synapse introduces 2 time scales: the short timescale of ordinary synaptic transmission and the longer timescale of NMDA activation. Again, there is evidence these 2 distinct timescales may have biological significance [Dur09]. Simply by setting a different delay to the deferred event, the processor can map input and output processes to entirely different time domains. In large systems, there might be many such domains, residing on different processors; by communicating through events, the SpiNNaker system introduces an abstract time model that breaks large models into independent units, an architecture for greater scalability as well as biological realism.

8.3 Scaling to Very Large Systems

SpiNNaker and most hardware neural systems under development are designed to model very large systems, containing a substantial fraction of a full human brain. It thus comes as no surprise that scalability has become one of the critical topics in neural networks. Systems that worked acceptably at 1000 or even 10,000 neurons are completely irrelevant at 10,000,000 neurons. It might be feasible to hand-design a 10^4 neuron network but a 10^7 neuron network requires reexamining the implementation model. Previous approaches to neural network design have tended to take the simulator or underlying hardware platform as a given. By contrast, the SpiNNaker neuromimetic platform provides almost complete model flexibility but does not indicate self-evidently how to implement them. This places much greater emphasis on pre-instantiation verification, and consideration of the hardware characteristics as an integral part of the design process. The HDL design-simulate-synthesize-implement flow is in this situation more appropriate, and has been the driving factor behind its adoption and adaptation for the SpiNNaker software model. HDL tools have proven effective at scaling designs from 1K gates to 1M+ gates; if the same scalability is to occur with SpiNNaker systems, it is necessary to consider both its hardware limitations and the broader question of how to scale neural models.

8.3.1 Hardware Scaling Challenges

The investigations confirm that it is *communications* rather than *algorithmic details* that generate most of the challenges in implementing neural networks on hardware. Notably, unexpected constraints or side effects of communications not only determined the final form of the MLP mapping but constituted a recurring theme during design and debugging. An illustrative example of this involves an interaction between the mapping and the communications.

SpiNNaker's asynchronous design provides no traffic management in the communications fabric. This can lead to local congestion, with potentially severe impact on performance and in extreme cases, on functionality: if packets arrive at a given processor faster than it can service them then it will start to miss packets altogether. Processors issue packets as soon as they are ready: thus if several packets become ready to send simultaneously or nearly so there will be a "burst" of local traffic. The "burstiness" of the traffic depends on the ratio of time spent in the packet-receiving service routine and the time spent in the packet-issuing background process. If the time spent in packet reception is large compared to the time spent in the background task there is a strong risk of local congestion. This is precisely the situation for the MLP model in the sum processors, since the accumulate process takes only 3 instructions per update. To minimise the risk of congestion the effective time in the background task needs to be "stretched". The easiest and most productive way to do this is to give the sum processor more tasks to do. In particular, sum processors can also assume the Monitor processor rôle, running the sum as an event-driven service and the monitor as the background task. In the test chip in particular, containing 2 processors per die, it is a critical technique that makes it possible to implement weight and sum units on the same chip while retaining Monitor functionality.

Both the MLP model and spiking models break down catastrophically if the packet traffic overwhelms the processors' ability to keep up. In the spiking model, this occurs when the neurons become excessively bursty. In the MLP model, this occurs when any one of the 3 component processes becomes disproportionately faster (i.e. simpler) than the others. Large network sizes exacerbate the problem in both cases. This issue appears to be fundamental in a truly concurrent processing system where individual processors operate asynchronously and independently. Finding effective ways to manage the problem, which does not arise in synchronous systems because of the predictable input timing relationships, is a critical future research topic.

8.3.2 Neural Model Scaling Challenges

The problem of scaling neural models to large sizes is, fundamentally, one of levels of abstraction: how much detail is necessary in the neural model? Biological neuroscientists, in particular, often express deep concern over oversimplifying the neural model in the interests of larger model size [Dur09]. While SpiNNaker in some ways side-steps this problem in allowing multiple levels of abstraction on the same system, it is nonetheless important to establish some kind of design methodology for scalability. Implementation of the LIF model suggests a solution:

Globally Abstract, Locally Detailed (GALD) modelling.

Any research project must of necessity make a definite choice as to the scope of the research: *what* is being investigated. The LIF model, on SpiNNaker or virtually any other programmable system, can support more complex synaptic models than other spiking models such as the Izhikevich model or conductance-based models, simply because it has fewer instructions to perform. This in itself is a strong reason to make it the model of choice for large-scale studies of synaptic dynamics, but there is another and equally powerful motivation: ease of analysis. Unsurprisingly, the majority of investigations into new synaptic models have used the LIF neuron so as not to introduce too many experimental variables simultaneously. It is ideal because it is very simple, exhaustively analysed, and does not introduce complexities that might obscure the effects of a given synaptic model.

The GALD procedure, then, is straightforward: choose an overall abstraction that is, or can be, analysed and characterised completely. This abstraction should be the simplest model that reproduces the desired effects for the focus of study, and it should be sufficiently debugged so as not to introduce unknown errors. Then, for the particular component or behaviour of interest, generate an appropriate, detailed model, which may be more experimental, unproven, or prototypical. SpiNNaker's ability to support multiple heterogeneous models makes it possible to implement both on the same system, and also permits insertion of different detailed models into the same large-scale abstraction. Using the LIF model as a base neuron for a model containing both NMDA and AMPA/GABA synapses demonstrates the GALD model in use.

How to manage concurrency in very large models with potentially billions of parallel processes is challenging regardless of the hardware platform. The problem is one of peak local resource utilisation. In a synchronous system, it is easy to predict processing load and interprocess traffic, since all updates occur in "lock-step". By contrast, in an event-driven system, there is no way to make such predictions beyond pessimistic upper-bounds calculations, and consequently it becomes vital to distribute processing and communications load uniformly. There are 2 basic approaches: changing the spatial processing distribution through the mapping, or changing the temporal processing distribution through scheduling.

In some networks the model architecture contains natural hierarchies that suggest an obvious mapping. In the attention-control network it is straightforward to map each neural "box" to a processor, and equally straightforward to scale it by increasing the size of a box to a chip, or to several adjacent chips. However, in the MLP model the "obvious" mapping of each unit to a specific processor, grouped by layer, performs poorly because it effectively doubles the communications traffic. There is therefore a two-stage mapping process: determine efficient mapping primitives, then scale these hierarchically. The central rôle of the neural library is clear: it contains the mapping primitives, which can be carefully hand-optimised and passed into automatic network "synthesis" tools that generate the hierarchical structures.

The deferred-event model is the principal vehicle for temporal processing distribution. By rescheduling processes to their required completion time, the deferred-event model can spread out the processing of a series of events occurring simultaneously; it does not have to handle them immediately and thus can reduce transient processor congestion. With its complex dynamic model, the NMDA synapse employs deferred-event processing aggressively. As the data shows (fig. 7.8), however, the deferred-event model has limitations that can become a factor in very large systems. The model relies on large differences of scale between electronic time and model time, because there is a fixed overhead associated with deferring the update. This is problematic if the input event rate is very high: if the frequency of events times the time to update for a single event exceeds the mean input-to-output delay in the model, the simulation will slow down. In large systems this becomes a greater danger because the larger number of potential inputs means the event frequency has a higher maximum. Additionally, the deferred-event model, or indeed any model of temporal processing distribution, increases memory requirements because of the need to store state information. For neurons, this may not be crippling at large scales because the amount of state will scale linearly; however, for synapses the state information can scale quadratically. It is vital, therefore, to identify methods like the STDP timestamp method, that can store state information associatively by an entire matrix row or column. The NMDA synapse, for example, aggregates NMDA synapses into a bin stored per-neuron, and exploits the linear kinetic of the NMDA response to precompute future NMDA activation, rather than store activations on a per-synapse basis. Temporal processing distribution is a useful and proven technique, however, it also demonstrates the general scaling issues with asynchronous event-driven neural processing that do not occur in synchronous systems, have as a result not been considered systematically, and need continued future research to develop formal theories with good predictive power.

8.4 Full-Brain Models? Computational Cognitive Neuroscience

Considerable work remains to be done in the area of neural development tools, both on SpiNNaker and more generally for neural hardware simulation. The implementation to date has focussed on developing a working basic system. Future work will aim in the first place at extending the model to larger and more complex systems, and in the second on detailed investigation of optimal mappings and translation of real neural architectures onto the physical hardware. For larger systems statistical description models as well as formal theories for neural network model design may be necessary. There remains also an open question of verification in a GALS system such as SpiNNaker: with nondeterministic timing behaviour, exact replication of the output from simulation is both impossible and irrelevant. It is important to develop meaningful test criteria for the finished device, focussing on replication of timing in the neural model independent of system-level timing.

Thus far the largest model size it has been possible to consider is a system implementing approximately 4500 neurons, a function of memory limitations in the simulation environment. What impact the increased model size possible through hardware simulation on the full-scale device will have on performance, mapping, and congestion concerns is still unknown. Particularly critical will be performance evaluation of the NoC under fully-loaded conditions, to generate hardware-realistic upper bounds on numbers of neurons and synapses per chip, and number of updates per second. One important research direction is topologies, memory update methods, and network protocols for managing densely connected networks such as cerebellar Purkinje cells which may require distributed implementations spread over several processors on a chip or several chips in a system, similar to the MLP model.

Scaling to larger networks will involve additional future work. Developing a high-level mapping tool to automate the “synthesis” process is a priority. With *very* large networks, particularly, the mapping problem is nontrivial and one solution under investigation is to load a “boot” network onto SpiNNaker to compute the mapping, which then in turn loads the “applications” network according to the mapping it has determined [BLP+09]. This exploits both the known capabilities of neural networks to solve complex mapping problems and the parallelism of the SpiNNaker hardware. More immediately the automation will focus on simple processor mappings to generate the routing tables and the data structures.

Likewise, it will be useful to automate data gathering and I/O, particularly during the simulation. One finding of the research was that dedicating a packet type to output run-time data, for debugging or data analysis, would be an extremely useful extension allowing greater internal visibility. The full-scale chip incorporates this design change and I/O routines that use this functionality are being added to the library. Work is ongoing on expanding the neural library with additional neural models, including extensions to other non-spiking representations. These library models will in time integrate into a PyNN-based user development environment that allows the modeller to implement the neural model with a high-level graphical or text description and use the automated generation flow to instantiate it upon SpiNNaker.

More work remains on identifying efficient methods to implement the weight updating. The deferred-event model provides a method to manage the update problem, but as the research shows, with relatively complex models such as the NMDA synapse, high activity in highly connected areas can severely reduce the number of synapses that SpiNNaker can model in real-time. The synapse channel also exists in the context of a system whose global connectivity, through the routing table, is in principle dynamically reconfigurable at run time. How synapse updates may be maintained consistent in such an environment remains an open issue. Meanwhile, models exhibiting completely different update dynamics, such as the MLP model, require methods to correlate weight update times with those synapses whose values change in a given update epoch. How learning should be scheduled in a large, complex system will therefore be a critical future research focus.

Now that the SpiNNaker chip is available, testing more models on the physical hardware is an obvious priority, particularly larger and more complex models that are impractical to simulate purely in software. Such models could simulate major subsystems of the brain. An important part of these models will be hierarchical library components scalable across a wide range of model sizes. These components would implement "neural circuits", analogous to hardware macrocells in an HDL environment. By reworking the library into a series of C++ template classes, the template parameters can indicate the model type. This will further simplify future model development and provide a specification for third-party model building. A given macrocell can then use any lower-level neural model in the library, simply by changing the template parameters. This process is also hierarchically scalable: macrocells in turn can form template parameters for larger units, ultimately leading to full-brain models. Clearly such components will improve automation capabilities but even more importantly, as abstract neural blocks, they could offer important insights into mesoscale brain structure and dynamics. Library development is thus a form of computational cognitive neuroscience, whose aim is to offer reasonable hypotheses about brain function by replicating behaviours using abstract neural networks with at least some biological plausibility.

The computational cognitive neuroscience approach will develop a series of models of ever-increasing hierarchical scale: first, neuron models and simple "subcircuits" (this is what has been achieved to date); next, neural "circuits", i.e. macrocells; then neural blocks replicating complete brain subsystems; finally full-scale brain modelling. One way to verify biological correspondence, particularly as the hierarchical scale goes beyond what it is possible to compare directly against neural recordings, would be to create a working network that successfully replicates certain behaviours, then alter parameters or connectivity in a way that generates pathologies similar to those observed in real clinical studies. If these variations can be correlated against FMRI or other active brain-imaging techniques, it will be possible to make a direct identification of brain regions with function, and potentially quantify the effects of parameter variation.

This is an important first step towards the development of objective formulae for calculating network parameters and structure. For example, in the study of connectivity and gain values in an MLP model, the data analysis is only preliminary. Clearly there is scope for further development of the gain/connectivity model. An important future topic is the development of methods for automated parameter extraction from raw input data: some means to normalise data characteristics so that models can use them directly in parameter calculation. Little work has been done on formal methods to compute connectivity, gain, or for that matter any other parameter of neural networks, whose design remains partially empirical; it seems timely that this should change. The increasing interest in larger neural networks inevitably means that empirical methods cannot in any case continue: what is adequate for optimising small networks of tens or hundreds of neurons becomes completely infeasible at the scale of tens of thousands or millions where fully automatic methods are essential. Perhaps the most enduring aspect of the work to create a neural library architecture, therefore, will be in providing a framework for abstraction: a process by which the *principles*, not just the *phenomenology* of biological neural networks can be uncovered.

8.5 Summary

Event-driven parallel systems like SpiNNaker represent an attempt to break the "monopoly" on models of computation the sequential digital model has held, by substituting another known-working model: that of the brain. Such very different systems demand a completely different software model; indeed, what constitutes *software* or whether the term itself is relevant at all needs reassessment. Whatever the term used, building a development system for a platform like SpiNNaker exposes assumptions taken virtually as axiomatic in computing and replaces them with different considerations having less to do with process determinism and more to do with behavioural feasibility:

Predesigned model libraries provide broad scope for a universal neural chip to function as a practical modelling tool

This is the first and most basic finding: it is possible to build libraries that support several different neural models, and expect them to reproduce neural behaviour. It has been possible to devise a general event-driven abstraction (the function pipeline), suitable for almost any type of neural network, that maps easily onto the hardware.

Event-driven computation improves process scheduling flexibility

The strongest justification for the move to event-driven computing is its facility for temporal abstraction. At its simplest, it improves scalability by limiting the number of active processes to the number receiving current events. More sophisticatedly, the deferred-event model permits arbitrary process reordering. In the limit, some neural dynamics may *only* be modellable with an event-driven model that does not require arbitrarily fine divisions of time to be explicit.

Pre-instantiation simulation is a prerequisite for working neural models

The event-driven model differs sharply from the sequential model in the need for simulation. Overall system behaviour is not predictable in the presence of asynchronous events, and meanwhile, the hardware provides only limited internal visibility. This makes event-driven debugging hard. Event-driven software-based simulators give the additional visibility necessary to implement models in the confidence that they will work on the hardware. The design-code-run-debug development cycle of traditional software thus changes to a design-build-simulate-modify-run-debug pattern.

Neural networks present an important tradeoff between representational richness and process efficiency

Choice of representation is critical for accurate neural modelling. On the one hand, an astute choice may significantly improve hardware processing. On the other, the process of abstraction involves a a tradeoff between representation and performance. While abstract models are necessary to create scalable neural models on real hardware, these are not necessarily unrealistic. Biological neural networks may, indeed, exploit the representational tradeoff, as in the connectivity-gain experiments which reveal real processing advantages from reduced representational capability. While in conventional processing precision tends to be a constraint, here, better precision does not necessarily mean a more accurate model: it is better to think of it as a system parameter.

Communications, rather than computation, is the limiting factor in neuromimetic systems

It is fairly easy, by replicating processors, to increase the computing power in an "embarrassingly parallel" application like neural networks, but the communications then scale with order n^2 in the number of processors, a much more difficult task. Event-driven systems can exacerbate this problem if comms traffic is "bursty": because the asynchronous event-driven architecture does not provide any intrinsic pacing of dataflow, events could happen in quick succession, overwhelming local communications capacity. Thus hardware limitations tend to become significant abruptly rather than gracefully. Event-driven neural models thus need to treat communications capacity as a constraint.

A library is only as good as the models it supports

The library-based development model relies on the availability of suitable preimplemented neural models. This is clearly a limitation from the user point of view. But current development tools have poor support for event-driven models, thus users, and even programmers, will probably encounter difficulties in trying to implement neural networks on-chip by a "direct-coding" approach. This makes the library model a logical step. Dedicated developers familiar with the hardware can focus on implementing efficient base models while users can concentrate on higher-level network design. Obviously, this emphasizes the need for further third-party SpiNNaker library development, just as the models thus far created form a basic "reference set".

Chapter 9

Conclusions: What Model for Neural Network Development?

Neural component libraries have proven to be an effective, if not indeed an indispensable, tool to solve the question of instantiating efficient, functional neural models on an application-specific neural processor. What significance does this have for neural research?

A universal neural chip can address the central problem in neural network research: what is the model of computation?

Chapter 1 stated this as the central thesis. The purpose of neural modelling is, and has been, to understand the neural model of computation, whether as it applies directly in biological behaviour, or in providing useful applications to general-purpose computing. This universal chip, a configurable device somewhat like an FPGA but more specifically designed to implement a neural model, can answer this question because it can be a platform for model abstraction. However, to be useful, it needs design tools, and here a library of neural functions that could be used as "drop-in" components addresses the first and most basic need: design automation.

A configurable, AER-based device relieves problems of scalability and model support that have plagued previous generations of neural devices.

Reviewing progress in the field, as Chapter 2 does, it appears there have been two main barriers to widespread neural hardware adoption, scalability and model support. Early devices had a tendency to attract only limited adoption because of a hardwired model structure, while later generations such as FPGA's uncovered difficult scaling barriers, particularly in interconnect density and power requirements. What has been needed is some sort of scalable standard, and here Address-Event Representation (AER) has emerged as a standard that overcomes the interconnect problem by allowing almost complete virtualisation of the topology. By combining this interconnect with an array of general-purpose, low-power processors it is possible to achieve a scalable, arbitrary-model neural system: the concept behind SpiNNaker.

The "neuromimetic" architecture: generic processing blocks embedded in a network-like interconnect, is a suitable next-generation neural architecture

Chapter 3 detailed the architecture of the SpiNNaker chip, as an example of the neuromimetic architecture. Two key architectural points make SpiNNaker a practical tool: an array of parallel ARM968 processors, that as functionally isolated computing nodes, achieve the massive parallelism that is central to neural computations, and a routable asynchronous packet-switched interconnect, that achieves the other pre-eminent feature of "real" neural networks: reconfigurable connectivity. This chip serves as a model for future neural systems: configurable in function, yet neurally-optimised in structure.

A library-based system with standard interfaces and components can provide the necessary framework for automation of event-driven neural network development

Chapter 4 motivated the creation of a library of neural functions, using a 3-level model to provide successive levels of abstraction to the user, from signal-level hardware visibility to behavioural neural model descriptions. This library handles the difficult work of translating hardware event representations into neural models, so that it is possible to define the model entirely in the high-level environment and have automated tools transform it into a SpiNNaker realisation. The architecture borrows its model from the hardware description language community, using a universal function pipeline to represent an abstraction of both the hardware components and the neural components. This library forms a "translation layer" between the physical hardware and the virtual model.

The event-driven model is usable both for spiking and nonspiking networks

Chapter 5 provided a concrete demonstration of the universal power of the event- driven libraries, by showing how to implement 3 different neural networks with entirely different structure and dynamics. Two spiking models illustrate how it is possible to use different levels of dynamic detail to investigate different aspects of neural computation. A third, multilayer perceptron model, provides an example of how to implement a completely non-spiking neural network in an event-driven environment. Successful implementation of these 3 models provides strong evidence of the universal power of the library approach.

The ability to model heterogeneous neural networks in a neuromimetic chip like SpiNNaker gives scope for exploring models' computational properties systematically

Having built the models, chapter 6 demonstrated through experiment how to use the models to identify neural networks' computational properties. It also indicates how such models can characterise the performance of the hardware and identify its limitations. Verification both at the hardware level and the model level is a critical part of neural design, and these results show that the verification process can also be used before hardware implementation to refine the models themselves.

Event-driven neural hardware improves scalability as well as speed

Chapter 7 investigated larger networks on the physical hardware. Ability to implement networks of reasonable size, even on a single chip, together with demonstration of effective multi-chip simulation, shows how the the event-driven model, by simplifying the communications, allows larger networks. Integration of SpiNNaker with external hardware in fully-functional real-world scenario both illustrates the value of the AER standard and achieves functionality that would have been challenging with conventional hardware of similar scale.

The universal event-driven neural architecture introduces challenges in communications management but supports model abstraction relatively easily

Chapter 8 discussed the immediate implications of the experiments. It is clear that implementing multiple heterogeneous models presents no particular problems *per se*, but that efficient packet management is essential. SpiNNaker fails abruptly when packet traffic exceeds a critical threshold, and this puts definite limits on activity levels, particularly with complex dynamics like NMDA synapses. SpiNNaker's completely virtual architecture, however, suggests a solution: scalable levels of abstraction in both a structural and temporal domain, permitting local detail where necessary and global simplification where possible.

Universal neural chips like SpiNNaker enable a structured future for neural modelling

> The current chapter turns to the future: what impact SpiNNaker and its library- based development model can have for ongoing neural research. Short-term, it will by force of necessity allow for powerful, general methods of model design automation and data analysis. Equally, SpiNNaker is a useful prototype hardware architecture that could be a standard for future chip designs, possibly incorporating more advanced functionality. More enduringly, the hope is that with SpiNNaker libraries it will be possible to uncover formal theories of neural computation that will permit a structured approach to neural network research.

These observations indicate that the SpiNNaker neuromimetic system is a new architecture for computation, which may even have applications beyond the neural domain for which it was designed. However, it also hints at significant changes in thinking that will be required in order to make the event-driven model a viable alternative to mainstream synchronous processing and simulation. Combined with a certain resistance from the biological modelling community, who expect either molecular-level fidelity or formal proof of the biological validity of the model, this means that the library system of neural components for SpiNNaker is still merely an entry point: a way of attracting interest in an important alternative neural hardware architecture. Indeed, even the computational community will likely be guarded in their adoption, until it is possible to make compelling demonstrations of SpiNNaker's capability to model a large-scale neural network involving heterogeneous model elements with greater speed or size that that achievable with conventional technology. What the library architecture introduces is a pragmatic roadmap for reaching the goal of biologically realistic large-scale neural simulation, *regardless of hardware platform*, as opposed to the ad-hoc, implementation-coupled method that currently prevails in the neuromorphic community. What does this roadmap look like?

9.1 Now: Library-Driven Neural Hardware Instantiation

Libraries of hardware routines are essential to providing a usable platform for temporal abstraction. Much of the driving motivation behind creating a library-based system for SpiNNaker has been that existing tools provide very poor support, if any at all, for asynchronous event-driven processes. The most vexing difficulties in creating libraries themselves, indeed, have been consistently those of trying to get industry-standard tools to generate efficient event-driven implementations. From the user perspective, a model library is the only practical way to generate asynchronous neural models: it is not realistic to expect users to struggle through tool sets whose design incorporates the synchronous model of computation at the outset. While the library model brings benefits to spatial abstraction that are *useful* in respect of creating a standard, its definitions of abstract-time models systematise the approach into a single architecture that gives neural network modelling a *real* chance of becoming a standard.

As an example, the LIF model is an important reference model for developing new synaptic models, and has been instrumental in creating and refining the function pipeline, as a base standard for neural and synaptic libraries. Reworking the library into a series of C++ template classes, where the template parameters indicate the model type, will further simplify future model development and provide a specification for third-party model building. Such a system should form an effective, scalable tool chain to develop a demonstration network for a multichip SpiNNaker system.

By contrast, the unusual nature of the challenges encountered in implementing the MLP model begs the question: are nonspiking models a “force-fit” for SpiNNaker, and if so, what hardware architectures are appropriate for MLP-style networks? The past history of hardware for MLP and other such neural networks is not encouraging, at least not to judge by commercial success [Omo00]. Both fixed-model and configurable FPGA designs, after a flurry of early interest, quietly faded away, an outcome that perhaps was to be expected given synchronous design, which would put them squarely in competition with existing conventional synchronous processors. The conventional processor has the advantage of continuous industry improvement and refinement in a technology firmly in the volume production sector, and with multiprocessor architectures now becoming mainstream, further development in synchronous neural hardware seems unattractive. This leaves processors such as SpiNNaker as the best, possibly the only, significant alternative. But more than simply a different architectural model offering different tradeoffs that could be useful in particular cases, SpiNNaker offers a distinguishing advantage: universal large-scale modelling capability.

The MLP is a popular and well-established neural model that will probably remain significant for a long time to come, but in addition, dynamic models with spiking behaviour are now popular. Such models are even more difficult than the MLP to adapt to conventional hardware and thus have even more powerful motivation to move to an on-chip solution. Thus an architecture that can provide effective acceleration for both classes of network has a better chance of success. With very large-scale models, furthermore, the limiting factor often turns out to be power; synchronous designs are power-hungry and hence encounter scaling barriers: as a point of reference the full-scale SpiNNaker chip consumes 1 W, a state-of-the-art Intel laptop processor such as the i7-640 18W, and a highly-parallel synchronous processor such as an NVIDIA GeForce 200 approximately 200W. The asynchronous, event-driven model thus may present challenges, but offers compelling advantages for large-scale neural modelling.

From a hardware/software architecture perspective, the question is, what is the ideal architecture for neural networks? On the hardware side, the research community seems to be converging on the AER packet-switched network: AER because it offers a lightweight, universal communications protocol with easy scalability, packet-switched because both bus and circuit-switched network designs far too quickly encounter interconnect density limitations [MMG+07]. Choice of processing elements remains more contentious, but a universal and familiar standard like the ARM makes it (relatively) easy to port existing models to SpiNNaker, giving more groups a logical basis to justify committing to the SpiNNaker platform.

The most important feature of a software architecture for universal neural modelling (particularly on dedicated hardware) would seem to be a modular structure to both "logical" and temporal objects. The concept of software modules is familiar from synchronous systems, leading to an object hierarchy, but in event-driven systems a similar hierarchical organisation of *event handlers* makes it easier to understand the process flow in neural networks. Individual models may vary, but the underlying event infrastructure can stay the same - and once again the research community is beginning to accept event-driven simulation as the most efficient and flexible modelling model. Events that can be used as flexibly as objects makes it possible to build a neural model by "connecting boxes", a more natural heuristic for neural design than language-based coding.

In a larger context, the function pipeline model developed for SpiNNaker may be a useful abstraction for neural hardware, regardless of platform. Creating the function pipeline was the result of an attempt to decompose the general form of neurodynamic state equations into platform-neutral components that hardware can typically implement easily. Digital hardware can readily implement memories to form variable retrieval and LUT stages, and both analogue and digital hardware have effective blocks for polynomial evaluation and interpolation. Both DSPs and various analogue blocks offer efficient computation of differential equations. Thus one could build a neural system in building-block fashion, by chaining together various components using AER signalling, allowing for the construction of hybrid systems in addition to integrated approaches like SpiNNaker. More than anything else, SpiNNaker is valuable as a test bed for hardware architectures for neural models.

9.2 Upcoming: Statistical Neural Synthesis

Making the hardware platform user-configurable rather than fixed-model introduces a new type of neural device whose architecture matches the requirements of large-scale experimental simulation, where the need to configure and test multiple and potentially heterogeneous neural network models within the same environment is critical. In this environment automated tools for development and simulation are essential. Design of an integrated hardware/software system like SpiNNaker provides a powerful model for neural network simulation: the hardware design flow of behavioural description, system synthesis, and concurrent simulation. Developing libraries provides the ability to leverage existing industry-standard tools and simulators so that it is unnecessary to develop a complete system from the ground up. This may well be critical for wider adoption of the architecture in the research, and possibly the industrial, community. However, scaling to very large systems will require similarly industrial-strength tools optimised for neural modelling.

The mapping model of synthesis from libraries is a proven technique that SpiNNaker can take straight from the hardware design community. However, traditional hardware synthesis requires an explicit, deterministic description of the system, at a relatively low level. Attempts to synthesize from high-level SystemC have, for the most part, yielded poor results [BF03], [LBB$^+$04]. So far, the current SpiNNaker software model has considered fixed, small-scale neural networks where it is practical to describe the network explicitly in terms of neurons, synapses, and connections. With such a description it is possible at least in principle to synthesize the network, because the tools can generate a neural component list and a netlist. However, for *very* large networks, this approach is infeasible. Under such conditions the most practical (and frequently-used) technique is to generate statistical models, describing populations of neurons and connectivity projections using probabilistic generator functions. It is perhaps just conceivable to imagine an intermediate tool that could run the generator functions and create a fixed netlist to input to the synthesizer, but the resulting files would undoubtedly be huge, to say nothing of the time to synthesize. A more creative approach is probably necessary.

One possible solution is to use the statistical power of neural network processing itself to generate the actual model network. Under this approach, SpiNNaker would first load a (fixed-mapping) "synthesizer" neural network whose job is to configure the running model network. This network might be similar to well-studied networks like the SOM that are effective in complex topological mapping problems. Such a tool would then auto-generate the running network itself from the high-level description. In this situation libraries are particularly valuable. The network could associate library modules with output units, or better, with specific patterns of activation in the output units. By preloading the available libraries into SDRAM before running the synthesizer network the tool could determine the network, configure the mapping, and distribute the code to processors without external interference, or possibly in an interactive process that provided the user with visibility on the state of progress of network loading. This makes best use of the hardware by taking advantage of its distributed-processing power: parallel neural synthesis on a parallel neural system [BLP+09].

9.3 Future: Formal Design of Neural Architectures

Sophisticated neural design and synthesis tools solve the immediate problem of automating very-large-scale neural network design for hardware systems such as SpiNNaker, but looking at the long-term future, there is a need for better formal theories of neural computation. This is, indeed, the basic purpose for which SpiNNaker was designed: to uncover the model of neural computation by modelling at user-definable levels of abstraction. When it becomes possible to feed back such theories into the design process, it will be possible to build even larger, more effective models using a systematic, formal design methodology. At the very largest scales, such models may be *necessary* in order to get intelligible results, and this is the ultimate goal of the neural library development programme: a hierarchical series of neural models incorporating formal scaling and connectivity principles that can be used to construct a neural network in the same way one might create a database. SpiNNaker is a near-ideal vehicle to realise this goal upon.

By implementing an event-driven model directly in hardware, SpiNNaker comes considerably closer to biological neural computation than clocked digital devices, while at the same time bringing into sharp relief the major differences from synchronous computation that place a much greater programming emphasis on the unpredictability of the flow of control. This important programming difference underscores the urgency for event-driven development tools, which at this point are scarce to nonexistent. Traditional languages like C assume a procedural, rather than an event-driven model. As a result, their support for event-driven processing is poor. This means, for practical purposes, that most critical event-driven routines must be written in assembly. This fact alone makes it clear why libraries are so important: even advanced users are likely to be unwilling to face writing in assembly. At best, however, the current library-based model can only be an interim solution; until the library routines themselves can be written in some abstract form third-party model development will probably remain limited. This requires an absolutely fundamental rethink of the methods used to define a process.

It is clear that most development tools today have an underlying synchronous assumption, which in addition to complicating development, tends to influence programmers' conceptual thinking - thus perpetuating the synchronous model. For example, even at a most basic level, the idea of programming in a *language* is fundamentally synchronous and sequential: it is confusing and difficult to express event dynamics in a language-like form. Possibly a development environment that moved away from a linguistic model towards graphically-orientated development, for example using Petri nets, might make it easier to develop for event-driven systems. If asynchronous dynamics is by definition a necessary feature of true parallel processing, perhaps the linguistic model is one reason why developing effective parallel programming tools has historically been difficult.

In the same way that the entire software model needs review, the hardware model for the neuromimetic architecture remains a work in progress. SpiNNaker involves various design compromises that future neuromimetic chips could improve upon. The interrupt mechanism in the ARM968 assumes a relatively slow interrupt rate. More forceful hardware could rectify this limitation. For example, if the vectored interrupt controller could *directly* vector the processor to the appropriate exception, bypassing the long and cumbersome entry point processing, interrupt rate could increase while narrowing critical time windows. Such a system might also have completely independent working memory ("register") banks for each exception, as well as a common area to pass data between exception modes without memory moves. Such features would be asking for data corruption in a synchronous model but become logical in the event-driven model.

The most obvious compromise, however, is the use of (locally) synchronous ARM968 processors. From the point of view of the entire system, local synchronicity is an implementation convenience, done almost entirely since existing off-the-shelf hardware tools and component libraries assume synchronous circuitry. In time it would also be ideal to move from a GALS system to a fully asynchronous system. The question of software and development tool support for modelling on such a device becomes even more critical than it is already: if an ideal device has a purely asynchronous internals, it needs purely asynchronous development tools as well. Ultimately, perhaps, such devices could incorporate analogue neuromorphic components and come with a mixed-signal descendant of the software model we have developed. Such a hybrid system would offer configurable processing with the speed and accuracy of analogue circuits where appropriate, and the density and flexibility of digital where advantageous. With such chips it would become meaningful again to speak of "direct implementation": the on-chip model could adapt to fit current research thinking, while the on-chip circuitry could implement the dynamic equations explicitly.

Getting to that point will almost certainly require richer and more biologically accurate abstractions of neural computation, grounded on a firmer theoretical footing. This is territory SpiNNaker is well-positioned to explore. One practical route to stronger neural processing theories is parameter space reduction. SpiNNaker allows models with, in principle, any level of biological realism. Thus it is perfectly possible to implement a (necessarily small or slow) network containing all known biological effects. Using known behaviours, it would then be possible to determine which effects are significant, simply by removing parameters or model dynamics one by one until the behaviour changes in a functionally meaningful way. One could also imagine a reverse approach: successively adding behaviours to a simplistic abstract model and determining which behavioural subtleties observed in real biology emerge with which additions. The process would thus build a neural function library with a hierarchy of modelling abstractions, and just as importantly, *rules* that the modeller can use to decide which model is situationally appropriate, or what analogue circuitry might be suitable in a mixed-signal chip design context. This system would be the future version, as much as SpiNNaker is the present version, of a neural network matching the development model and environment to the computational model.

Since their emergence in the 1950's, it seems neural networks have been seen mostly as an *algorithmic technique*, as opposed to an *architectural model*; if there is one central theme this work has attempted to show it is that the latter approach is now feasible. The idea of neural network as algorithm almost takes as a given the assumption that the computing platform is a synchronous, sequential processor; SpiNNaker changes the model entirely into an event-driven, parallel processor. The neural library, even in the prototypical state it is today, illustrates how to incorporate heterogeneous neural models into a single, event-driven abstraction, and there is reason to believe virtually any model will fit into the same library framework. In turn these libraries can form the building blocks for sophisticated automatic design tools, that will in the near future be able to take a high-level description of a neural network and turn it into a working instantiation on SpiNNaker - or possibly even other universal neural hardware. If the the long-term result of this is that better theories of neural computation, and a better understanding of the biology, emerge, then SpiNNaker will have achieved its purpose. At that time neural networks will be able to take their place alongside conventional sequential computers as a viable computing architecture, and neurobiologists will be able to decode the complex dynamics of the brain into clear and rational descriptions of thought processes. The future may hold nothing less than understanding, and *building*, human intelligence.

Bibliography

[AN00] L. F. Abbott and S. B. Nelson. Synaptic plasticity: taming the beast. *Nature Neuroscience*, 3(11s):1178–1183, November 2000.

[APR$^+$96] M. Anguita, F. J. Pelayo, E. Ros, D. Palomar, and A. Prieto. VLSI Implementations of CNNs for Image Processing and Vision Tasks: Single and Multiple Chip Approaches. In *Proc. 4th IEEE Int'l Wkshop. Cellular Neural Networks and Their Applications (CNNA-96)*, pages 479–484, 1996.

[ASAW06] F. Aminian, E. D. Suarez, M. Aminian, and D. T. Walz. Forecasting Economic Data with Neural Networks. *Computational Economics*, 28(1):71–78, August 2006.

[ASHJ04] M. Arnold, T. Sejnowski, D. Hammerstrom, and M. Jabri. Neural Systems Integration. *Neurocomputing*, 58–60:1123–1168, June 2004.

[Ban01] A Banerjee. On the Phase-Space Dynamics of Systems of Spiking Neurons I: Model and Experiments. *Neural Computation*, 13(1):161–193, January 2001.

[BC88] C. H. Bailey and M. Chen. Long-term memory in *aplysia* modulates the total number of varicosities of single identified sensory neurons. *Proc. Nat. Acad. Sci. USA*, 85(7):2373–2377, April 1988.

[BD08] D. W. Barr and P. Dudek. A Cellular Processor Array Simulation and Hardware Prototyping Tool. In *Proc. 11th Int'l Wkshp. Cellular Neural Networks and Their Applications (CNNA2008)*, pages 213–218, 2008.

[BDCG07] D. R. W. Barr, P. Dudek, J. M. Chambers, and K. Gurney. Implementation of multilayer integrator networks on a cellular processor array. In *Proc. 2007 Int'l Joint Conf. Neural Networks (IJCNN2007)*, pages 349–352, 2007.

[BF03] F. Bruschi and F. Ferrandi. Synthesis of Complex Control Structures From Behavioral SystemC Models. In *Proc. 2003 Design, Automation, and Test in Europe Conf. and Exhibition (DATE 2003)*, pages 112–117, 2003.

[BGM+07] D. Brüderle, A. Grübl, K. Meier, E. Müller, and J. Schemmel. A Software Framework for Tuning the Dynamics of Neuromorphic Silicon Towards Biology. In *Proc. 2007 Int'l Wkshp on Artificial Neural Networks (IWANN 2007)*, pages 479–486, 2007.

[BHS98] J.-L. Beuchat, J.-O. Haenni, and E. Sanchez. Hardware Reconfigurable Neural Networks. In *Proc. 12th Int'l Parallel Proc. Symp. and 9th Symp. Parallel and Distributed Processing (10 IPPS/SPDP'98)*, pages 91–98, 1998.

[BI07] C. Bartolozzi and G. Indiveri. Synaptic Dynamics in Analog VLSI. *Neural Computation*, 19(10):2581–2603, October 2007.

[BK06] F. Bernhard and R. Keriven. Spiking Neurons on GPUs. In *Proc. 6th Int'l Conf. Computational Science (ICCS 2006)*, pages 236–243, 2006.

[BLP+09] A. Brown, D. Lester, L. Plana, S. Furber, and P. Wilson. SpiNNaker: The design automation problem. In *Proc. 2008 Int'l Conf. Neural Information Processing (ICONIP 2008)*, pages 1049–1056. Springer-Verlag, 2009.

[BMD+09] D. Brüderle, E. Müller, A. Davison, E. Muller, J. Schemmel, and K. Meier. Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system. *Frontiers in Neuroinformatics*, 3(17), June 2009.

[Boa00] K. A. Boahen. Point-to-Point Connectivity Between Neuromorphic Chips Using Address Events. *IEEE Trans. Circuits and*

Systems 2: Analog and Digital Signal Processing, 47(5):416–434, May 2000.

[BÖD$^+$99] İ Bayraktaroğlu, A. S. Öğrenci, G. Dündar, S. Balkir, and E. Alpaydin. ANNSyS: an Analog Neural Network Synthesis System. *Neural Networks*, 12(2):325–338, March 1999.

[BP98] Guoqiang Bi and Muming Poo. Synaptic Modifications in Cultured Hippocampal Neurons: Dependence on Spike Timing, Synaptic Strength, and Postsynaptic Cell Type. *J. Neurosci.*, 18(24):10464–10472, December 1998.

[BPS10] M. A. Bhuiyan, V. K. Pallipuram, and M. C. Smith. Acceleration of Spiking Neural Networks in Emerging Multi-core and GPU Architectures. In *Proc. 2010th IEEE Int'l Symp. on Parallel and Distributed Processing Wkshps. (IPDPSw 2010)*, pages 1–8, 2010.

[BRE94] J. A. Barby, S. E. Rehan, and M. I. Elmasry. AHDL modelling to support top-down design of mixed-signal ASICs. In *Proc. 7th IEEE Int'l ASIC Conf. and Exhibit*, pages 166–169, 1994.

[CAD07] L. Calcroft, R. Adams, and N. Davey. Efficient architectures for sparsely-connected high capacity associative memory models. *Connection Science*, 19(2):163–181, June 2007.

[Cau96] G. Cauwenberghs. An Analog VLSI Recurrent Neural Network Learning a Continuous-Time Trajectory. *IEEE Trans. Neural Networks*, 7(2):346–361, March 1996.

[CEVB97] S. M. Crook, G. B. Ermentrout, M. C. Vanier, and J. M. Bower. The Role of Axonal Delay in the Synchronization of Networks of Coupled Cortical Oscillators. *J. Computational Neuroscience*, 4(2):161–172, April 1997.

[CHC06] B. L. Chen, D. H. Hall, and D. B. Chklovskii. Wiring optimization can relate neuronal structure and function. *Proc. Nat. Acad. of Sciences of the USA*, 103(12):4723–4728, March 2006.

[CJ92] Chwan-Hwa Wu and Jyun-Hwei Tsai. Concurrent Asynchronous Learning Algorithms for Masssively Parallel Recurrent Neural Networks. *J. Parallel and Distributed Computing*, 14(3):345–353, March 1992.

[CL05] E. Cheong and J. Liu. galsC: A Language for Event-Driven Embedded Systems. In *Proc. Design, Automation and Test in Europe 2005 (DATE'05)*, pages 1050–1055, 2005.

[CMAJSGLB08] L. Camuñas-Mesa, A. Acosta-Jiménez, T. Serrano-Gotarredona, and B. Linares-Barranco. Fully Digital AER Convolution Chip for Vision Processing. In *Proc. 2008 IEEE Int'l Symp. Circuits and Systems (ISCAS2008)*, pages 652–655, 2008.

[CRT07] A. V. Chizhov, S. Rodrigues, and J. R. Terry. A comparative analysis of a firing-rate model and a conductance-based neural population model. *Phys. Letters A*, 369(1–2):31–36, September 2007.

[CSF04] G. La Camera, W. Senn, and S. Fusi. Comparison between networks of conductance- and current-driven neurons: stationary spike rates and subthreshold depolarization. *Neurocomputing*, 58–60:253–258, June 2004.

[CWL$^+$07] E. Chicca, A. M. Whatley, P. Lichtsteiner, V. Dante, T. Delbruck, P. del Giudice, R. J. Douglas, and G. Indiveri. A Multichip Pulse-Based Neuromorphic Infrastructure and Its Application to a Model of Orientation Sensitivity. *IEEE Trans. Circuits and Systems*, 54(5):981–993, May 2007.

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

[DAM04] F. M. Dias, A. Antunes, and A. M. Mota. Artificial neural networks: a review of commercial hardware. *Engineering Applications of Artificial Intelligence*, 17(8):945–952, December 2004.

[DBC06] P. Dong, G. L. Bilbro, and M.-Y. Chow. Implementation of Artificial Neural Network for Real Time Applications Using Field Programmable Analog Arrays. In *Proc. 2006 Int'l Joint Conf. Neural Networks (IJCNN2006)*, pages 1518–1524, 2006.

[DBE^{+}09] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in Neuroinformatics*, 2(11), January 2009.

[DCA06] N. Davey, L. Calcroft, and R. Adams. High capacity, small world associative memory models. *Connection Science*, 18(3):247–264, September 2006.

[DDT^{+}95] T. Daud, T. Duong, M. Tran, H. Langenbacher, and A. Thakoor. High Resolution Synaptic Weights and Hardware-in-the-Loop Learning. In *Proc. SPIE - Int'l Soc. Optical Engineering*, volume 2424, pages 489–500, 1995.

[Des97] A. Destexhe. Conductance-Based Integrate-and-Fire Models. *Neural Computation*, 9(3):503–514, April 1997.

[DHMM97] C. Diorio, P. Hasler, B. A. Minch, and C. A. Mead. A Floating-Gate MOS Learning Array With Locally Computed Weight Updates. *IEEE Trans. Electron Devices*, 44(12):2281–2289, December 1997.

[DLR02] G. Danese, F. Leporati, and S. Ramat. A Parallel Neural Processor for Real-Time Applications. *IEEE Micro*, 22(3):20–31, may-jun 2002.

[DT03] A. Delorme and S. Thorpe. SpikeNET: an event-driven simulation package for spiking neural networks. *Network: Computation in Neural Systems*, 14(4):613–627, November 2003.

[Dur09] D. Durstewitz. Implications of synaptic biophysics for recurrent network dynamics and active memory. *Neural Networks*, 22(8):1189–1200, October 2009.

[EH94a] J. Eldredge and B. Hutchings. Density Enhancement of a Neural Network Using FPGAs and Run-Time Reconfiguration. In *Proc. IEEE Wkshp. on FPGAs for Custom Computing Machines*, pages 180–188, 1994.

[EH94b] J. Eldredge and B. Hutchings. RRANN: the run-time reconfiguration artificial neural network. In *Proc. IEEE 1994 Custom Integrated Circuits Conf.*, pages 77–80, 1994.

[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.

[FGH06] E. Farquhar, C. Gordon, and P. Hasler. A Field Programmable Neural Array. In *Proc. 2006 IEEE Int'l Symp. Circuits and Systems (ISCAS2006)*, pages 4114–4117, 2006.

[FHK93] U. Frey, Y.-Y. Huang, and E. R. Kandel. Effects of cAMP Simulate a Late Stage of LTP in Hippocampal CA1 Neurons. *Science*, 260(5114):1561–1677, June 1993.

[FRSL09] A. K. Fidjeland, E. B. Roesch, M. P. Shanahan, and W. Luk. NeMo: A Platform for Neural Modelling of Spiking Neurons Using GPUs. In *Proc. 20th IEEE Int'l Conf. on Application-Specific Systems, Architectures and Processors*, pages 137–144, 2009.

[FSM08] J. Fieres, J. Schemmel, and K. Meier. Realizing biological spiking network models in a configurable wafer-scale hardware system. In *Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008)*, pages 969–976. IEEE Press, 2008.

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

[FWA88] B. Furman, J. White, and A. A. Abidi. CMOS Analog IC Implementing the Back Propagation Algorithm. *Neural Networks*, 1(Supplement 1):381, 1988.

[Gar87] S. C. J. Garth. A chipset for high speed simulation of neural network systems. In *Proc. IEEE 1st Int'l Conf. Neural Networks*, pages 443–452, 1987.

[GAS08] W. Van Geit, P. Achard, and E. De Schutter. Neurofitter: A parameter tuning package for a wide range of electrophysiological neuron models. *Frontiers in Neuroinformatics*, 1(5), November 2008.

[GB08] D. Goodman and R. Brette. Brian: a simulator for spiking neural networks in Python. *Frontiers in Neuroinformatics*, 2(5), November 2008.

[GE97] R. Gerndt and R. Ernst. An Event-Driven Multi-Threading Architecture for Embedded Systems. In *Proc. 5th Int'l Wkshp. Hardware/Software Codesign (CODES/CASHE'97)*, pages 29–33, 1997.

[GHJdV87] H. P. Graf, W. Hubbard, L. D. Jackel, and P. G. N. de Vegvar. A CMOS Associative Memory Chip. In *Proc. IEEE First Int'l Conf. on Neural Networks*, pages 461–468, 1987.

[GHMM09] B. Glackin, J. Harkin, T. M. McGinnity, and L. Maguire. A Hardware Accelerated Simulation Environment for Spiking Neural Networks. In *Proc. 5th Int'l Wkshp. on Reconfigurable Architectures, Tools and Applications. (ARC 2009)*, pages 336–341, 2009.

[GKvHW96] W. Gerstner, R. Kempter, J. L. van Hemmen, and H. Wagner. A Neuronal Learning Rule for Sub-millisecond Temporal Coding. *Nature*, 383(6595):76–78, Sep. 1996.

[GMM$^+$05] B. Glackin, T. M. McGinnity, L. Maguire, Q. X. Wu, , and A. Belatreche. A Novel Approach for the Implementation of Large-Scale Spiking Neural Networks on FPGA Hardware. In *Proc. 8th Int'l Work-Conf. on Artificial Neural Networks. (IWANN 2005)*, pages 552–563, 2005.

[GR94] S. Ghosh and D. L. Reilly. Credit Card Fraud Detection with a Neural-Network. In *Proc. 27th Hawaii Int'l Conf. on System Sciences*, pages 621–630, 1994.

[GSM94] M. Gshwind, V. Salapura, and O. Maischberger. RAN^{2}OM A Reconfigurable Neural Network Architecture Based on Bit Stream Arithmetic. In *Proc. 1994 Int'l Conf. Field Programmable Logic and Applications (FPL 1994)*, pages 1861–1864, 1994.

[GSW02] C. Grassman, T. Schönauer, and C. Wolff. PCNN neurocomputers - event driven and parallel architectures. In *Proc. 10th European Symp. on Artificial Neural Networks (ESANN'2002)*, pages 331–336, 2002.

[GT02] V. Gosasang and T. Tanprasert. Performance and Caching Issues in an Integration of Neural Net and Standard PC. In *Proc. 2002 Int'l Joint Conf. Neural Networks (IJCNN2002)*, pages 565–570, 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.

[HB06] K. M. Hynna and K. Boahen. Neuronal Ion-Channel Dynamics in Silicon. In *Proc. 2006 Int'l Symp. Circuits and Systems (ISCAS 2006)*, pages 3614–3617, 2006.

[HC97] M. L. Hines and N. T. Carnevale. The NEURON simulation environment. *Neural Computation*, 9(6):1179–1209, August 1997.

[HCR$^+$04] M. L. Hayashi, S.-Y. Choi, B. S. S. Rao, H.-Y. Jung, H.-Y. Lee, D. Zhang, S. Chattarji, A. Kirkwood, and S. Tonegawa. Altered Cortical Synaptic Morphology and Impaired Memory Consolidation in Forebrain-Specific Dominant-Negative PAK Transgenic Mice. *Neuron*, 42(5):773–787, June 2004.

[HF92] M. Hohfeld and S. E. Fahlman. Probabilistic rounding in neural network learning with limited precision. *Neurocomputing*, 4(6):291–299, December 1992.

[HHXW08] Hua Hu, Jing Huang, Jianguo Xing, and Wenlong Wang. Key Issues of FPGA Impelentations of Neural Networks. In *Proc. 2nd Int'l Symp. Intelligent Information Technology Application*, pages 259–263, 2008.

[Hil97] W. Hilberg. Neural networks in higher levels of abstraction. *Biological Cybernetics*, 76(1):23–40, January 1997.

[HM94] P. Hylander and J. Meador. Object Oriented VLSI Design Automation fo Pulse Coded Neural Networks. In *Proc. 1994 Int'l Joint Conf. Neural Networks (IJCNN1994)*, pages 1825–1829, 1994.

[HMH$^+$08a] J. Harkin, L. McDaid, S. Hall, T. Dowrick, and F. Morgan. Programmable Architectures for Large-scale Implementations of Spiking Neural Networks. In *Proc. 2008 IET Irish Signals and Systems Conf. (ISSC 2008)*, pages 374–379, 2008.

[HMH$^+$08b] J. Harkin, F. Morgan, S. Hall, P. Dudek, T. Dowrick, and L. McDaid. Reconfigurable platforms and the challenges for large-scale implementations of spiking neural networks. In *Proc. 2008 Int'l Conf. Field Programmable Logic and Applications (FPL 2008)*, pages 483–486, 2008.

[HMS08] M. L. Hines, H. Markram, and F. Schürmann. Fully implicit parallel simulation of single neurons. *J. Computational Neuroscience*, 25(3):439–448, December 2008.

[Hop82] J. J. Hopfield. Neural Networks and Physical Systems with Emergent Collective Computational Abilities. *Proc. National Academy of Sciences of the USA*, 79(8):2554–25588, April 1982.

[HP92] P. W. Hollis and J. J. Paulos. An Analog BiCMOS Hopfield Neuron. *Analog Integrated Circuits and Signal Processing*, 2(4):273–279, November 1992.

[HTCB89] M. Holler, S. Tam, H. Castro, and R. Benson. An Electrically Trainable Artificial Neural Network (ETANN) with 10240 “Floating Gate” Synapses. In *Proc. 1989 Int’l Joint Conf. Neural Networks (IJCNN1989)*, pages 191–196, 1989.

[HTT06] M. Hartley, N. Taylor, and J. Taylor. Understanding spike-time-dependent plasticity: A biologically motivated computational model. *Neurocomputing*, 69(16):2005–2016, July 2006.

[ICD06] G. Indiveri, E. Chicca, 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):211–221, January 2006.

[ICK96] P. Ienne, T. Cornu, and G. Kuhn. Special-Purpose Digital Hardare for Neural Networks: An Architectural Survey. *J. VLSI Signal Processing Systems*, 13(1):5–25, August 1996.

[IE08] E.M. Izhikevich and G. M. Edelman. Large-scale model of mammalian thalamocortical systems. *Proc. National Academy of Sciences of the USA*, 105(9):3593–3598, March 2008.

[IH09] E. M. Izhikevich and F.C. Hoppensteadt. Polychronous Wavefront Computations. *International Journal of Bifurcation and Chaos*, 19(5):1733–1739, May 2009.

[IWK99] G. Indiveri, A. M. Whatley, and J. Kramer. A Reconfigurable Neuromorphic VLSI Multi-Chip System Applied to Visual Motion Computation. In *Proc. 7th Int’l Conf. Microelectronics for Neural, Fuzzy, and Bio-Inspired Systems*, pages 37–44, 1999.

[Izh03] E.M. Izhikevich. Simple Model of Spiking Neurons. *IEEE Trans. Neural Networks*, 14:1569–1572, November 2003.

[Izh04] E. M. Izhikevich. Which Model to Use for Cortical Spiking Neurons. *IEEE Trans. Neural Networks*, 15(5):1063–1070, September 2004.

[Izh06] E.M. Izhikevich. Polychronization: Computation with Spikes. *Neural Computation*, 18(2), February 2006.

[JC09] Jihong Liu and Chengyuan Wang. A Survey of Neuromorphic Engineering. In *Proc. 2009 IEEE Circuits and Systems Conf. Testing and Diagnosis (ICTD'09)*, 2009.

[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. Neural Networks (IJCNN2008)*, pages 2812–2819, 2008.

[JG10] C. Joseph and A. Gupta. A Novel Hardware Efficient Digital Neural Network Architecture Implemented in 130nm Technology. In *Proc. 2nd Int'l Conf. Computer and Automation Engineering (ICCAE 2010)*, pages 82–87, 2010.

[JH92] M. James and Doan Hoang. Design of Low-Cost, Real-Time Simulation Systems for Large Neural Networks. *J. Parallel and Distributed Computing*, 14(3):221–235, March 1992.

[JL07] C. Johansson and A. Lansner. Towards cortex sized artificial neural systems. *Neural Networks*, 20(1):48–61, January 2007.

[JLK$^+$10] X. Jin, M. Lujan, M. M. Khan, L. A. Plana, A. D. Rast, S. Welbourne, and S. B. Furber. Efficient Parallel Implementation of a Multi-Layer Backpropagation Network on Torus-connected CMPs. In *Proc. 2010 ACM Conf. Computing Frontiers (CF'10)*, pages 89–90, 2010.

[JM95] G. Jackson and A. F. Murray. Competence Acquisition in an Autonomous Mobile Robot using Hardware Neural Techniques. In *Proc. 1995 Conf. Advances in Neural Information Processing Systems (NIPS 1995)*, pages 1031–1037, 1995.

[JPL98] Y.-J. Jang, C.-H. Park, and H.-S. Loo. A Programmable Digital Neuro-Processor Design with Dynamically Reconfigurable Pipeline/Parallel Architecture. In *Proc. 1998 Int'l Conf. Parallel and Distributed Systems*, pages 18–24, 1998.

[JRG$^+$10] X. Jin, A. D. Rast, F. Galluppi, S. Davies, and S. B. Furber. Implementing Spike-Timing-Dependent Plasticity on SpiNNaker

Neuromorphic Hardware. In *Proc. 2010 Int'l Joint Conf. on Neural Networks (IJCNN2010)*, pages 2302–2309, 2010.

[JS03] Jihan Zhu and P. Sutton. FPGA Implementations of Neural Networks - A Survey of a Decade of Progress. In *Proc. Int'l Conf. Field-Programmable Logic*, pages 1062–1066, 2003.

[JSR+97] A. Jahnke, T. Schönauer, U. Roth, K. Mohraz, and H. Klar. Simulation of Spiking Neural Networks on Different Hardware Platforms. In *Proc. 1997 Int'l Conf. Artificial Neural Networks (ICANN 1997)*, pages 1187–11192, 1997.

[KA95] J. V. Kennedy and J. Austin. A Parallel Architecture for Binary Neural Networks. In *Proc. 6th Int'l Conf. Microelectronics for Neural Networks, Evolutionary and Fuzzy Systems (MICRONEURO'97)*, pages 225–232, 1995.

[KBSM09] B. Kaplan, D. Brüderle, J. Schemmel, and K. Meier. High-Conductance States on a Neuromorphic Hardware System. In *Proc. 2009 Int'l Joint Conf. Neural Networks (IJCNN2009)*, pages 1524–1530, 2009.

[KdlTRJ06] Y. E. Krasteva, E. de la Torre, T. Riesgo, and D. Joly. Virtex II FPGA bitstream manipulation: Application to reconfiguration control systems. In *Proc. 2006 Int'l Conf. Field Programmable Logic and Applications (FPL 2006)*, pages 717–720, 2006.

[KDR07] T. Kaulmann, D. Dikmen, and U. Rückert. A Digital Framework for Pulse Coded Neural Network Hardware with Bit-Serial Operation. In *Proc. 7th Int'l. Conf. on Hybrid Intelligent Systems*, pages 302–307, 2007.

[Ker92] L. R. Kern. Design and development of a real-time neural processor using the intel 80170nx etann. In *Proc. 1992 Int'l Joint Conf. Neural Networks (IJCNN1992)*, pages 684–689, 1992.

[KGH09] T. J. Koickal, L. C. Gouveia, and A. Hamilton. A programmable spike-timing based circuit block for reconfigurable neuromorphic computing. *Neurocomputing*, 72(16–18):3609–3616, October 2009.

[KJFP07] M. 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 9–12, 2007.

[Kli90] C. C. Klimasauskas. Neural Networks and Image Processing. *Dr. Dobb's J. of Software Tools*, 15(4):77–82, 114, 116, April 1990.

[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. Neural Networks (IJCNN2008)*, pages 2849–2856, 2008.

[KM94] G. Kechriotis and E. S. Manolakos. Training fully recurrent neural networks on a ring transputer array. *Microprocessors and Microsystems*, 18(1):5–11, jan–feb 1994.

[Kön97] A. König. On Application Incentive and Constraints for Neural Network Hardware Development. In *Proc. 1997 Int'l Work-Conf. on Artificial and Natural Neural Networks (IWANN'97)*, pages 782–791, 1997.

[Kön98] A. König. Towards Actual Neural Coprocessors for Heterogeneous Embedded Systems. In *Proc. 4th Int'l. Conf. on Neural Information Processing (ICONIP 1997)*, pages 670–673, 1998.

[KRN$^+$10] M. M. Khan, A. D. Rast, J. Navaridas, X. Jin, L.A. Plana, M. Luján, S. Temple, C. Patterson, D. Richards, J. V. Woods, J. Miguel-Alonso, and S. B. Furber. Event-Driven Configuration of a Neural Network CMP System over an Homogeneous Interconnect Fabric. *Parallel Computing*, 36, November 2010.

[LBB$^+$04] D. Lettnin, A. Braun, M. Bodgan, J. Gerlach, and W. Rosenstiel. Synthesis of embedded SystemC design: A case study of digital neural networks. In *Proc. Design, Automation & Test in Europe Conf. & Exhibition (DATE'04)*, volume 3, pages 248–253, 2004.

[LBJMLBCB06] A. Linares-Barranco, G. Jiménez-Moreno, B. Linares-Barranco, and A. Civit-Barcells. On Algorithmic Rate-Coded AER Generation. *IEEE Trans. Neural Networks*, 17(3):771–788, May 2006.

[LBLBJMCB05] A. Linares-Barranco, B. Linares-Barranco, G. Jiménez-Moreno, and A. Civit-Barcells. AER Synthetic Generation in Hardware for Bio-inspired Spiking Systems. *Proc. SPIE*, 5839:103–110, 2005.

[LL95] C. S. Lindsey and T. Lindblad. Survey of neural network hardware. In *Proc. SPIE, Applications and Science of Artificial Neural Networks*, volume 2492, pages 1194–1205, 1995.

[LL10] E. Lehtonen and M. Laiho. CNN Using Memristors for Neighborhood Connections. In *Proc. 12th Int'l Wkshp. on Cellular Nanoscale Networks and Their Applications (CNNA2010)*, pages 1–4, 2010.

[LLW05] Z. Luo, H. Liu, and X. Wu. Artificial Neural Network Computation on Graphic Process Unit. In *Proc. 2005 Int'l Joint Conf. Neural Networks (IJCNN2005)*, pages 622–626, 2005.

[LMAB06] J. Lin, P. Merolla, J. Arthur, and K. Boahen. Programmable Connections in Neuromorphic Grids. In *Proc. 49th Midwest Symp. Circuits and Systems (MWSCAS 2006)*, pages 80–84, 2006.

[LS92] B. J. Lee and B. W. Sheu. General-Purpose Neural Chips with Electrically Programmable Synapses and Gain-Adjustable Neurons. *IEEE J. of Solid-State Circuits*, 27(9):1299–1302, September 1992.

[LSLG94] P. Lysaght, J. Stockwood, J. Law, and D. Girma. Artificial neural network implementation on a fine-grained FPGA. In *Proc. 4th Int'l Wkshop. Field-Programmable Logic and Applications (FPL '94)*, pages 421–431, 1994.

[LWM$^+$93] J. Lazzaro, J. Wawrzynek, M. Mahowald, M. Silviotti, and D. Gillespie. Silicon Auditory Processors as Computer Peripherals. *IEEE Trans. Neural Networks*, 4(3):523–528, May 1993.

[Maa01] W. Maass. On the relevance of time in neural computation and learning. *Theoretical Computer Science*, 261(1):157–178, June 2001.

[MAM05] M. Mirhassani, M. Ahmadi, and W. C. Miller. Design and Implementation of Novel Multi-Layer Mixed-Signal On-Chip Neural Networks. In *Proc. 48th IEEE Int'l Midwest Symp. Circuits and Systems*, pages 413–416, 2005.

[MCL$^+$06] M. Migliore, C. Cannia, W. W. Lytton, H. Markram, and M. L. Hines. Parallel network simulations with NEURON. *J. Computational Neuroscience*, 21(2):119–29, October 2006.

[ME08] V. Mahoney and I. Elhanany. A Backpropagation Neural Network Design Using Adder-Only Arithmetic. In *Proc. 51st Midwest Symp. on Circuits and Systems*, pages 894–897, 2008.

[MECG06] J. M. McGuinness, C. Egan, B. Christianson, and G. Gao. The Challenges of Efficient Code-Generation for Massively Parallel Architectures. In *Proc. 11th Asia-Pacific Conf. Advances in Computer Systems Architecture (ACSAC 2006)*, pages 416–422, 2006.

[MGS05] S. Marinai, M. Gori, and G. Soda. Artificial Neural Networks for Document Analysis and Recognition. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 27(1):23–35, January 2005.

[MJH$^+$03] N. Mehrtash, D. Jung, H.H. Hellmich, T. Schönauer, Vi Thanh Lu, and H. Klar. Synaptic Plasticity in Spiking Neural Networks (SP^{2}INN): a System Approach. *IEEE Trans. Neural Networks*, 14(5):980–992, September 2003.

[MLS04] S. McBader, P. Lee, and A Sartori. The Impact of Modern FPGA Architectures on Neural Hardware: A Case Study of the TOTEM Neural Processor. In *Proc. 2004 Int'l Joint Conf. Neural Networks (IJCNN2004)*, pages 3149–3154, 2004.

[MMG$^+$07] L.P. Maguire, T. M. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin. Challenges for large-scale implementations of spiking neural networks on FPGAs. *Neurocomputing*, 71, December 2007.

[MMP$^+$07] L. Melloni, C. Molina, M. Pena, D. Torres, W. Singer, and E. Rodriguez. Synchronization of Neural Activity across Cortical Areas Correlates with Conscious Perception. *J. Neurosci*, 27(11):2858–2865, March 2007.

[MR94] D. C. Marinescu and J. R. Rice. On the Scalability of Asynchronous Parallel Computations. *J. Parallel and Distributed Computing*, 22(3):538–546, September 1994.

[MS88] A. F. Murray and A. V. W. Smith. Asynchronous VLSI Neural Networks Using Pulse-Stream Arithmetic. *IEEE J. of Solid-State Circuits*, 23(3):688–697, June 1988.

[MS08] Y. Meng and B. Shi. Adaptive Gain Control for Spike-Based Map Communication in a Neuromorphic Vision System. *IEEE Trans. Neural Networks*, 19(6):1010–1021, June 2008.

[MSK98] B. Milner, L. R. Squire, and E. R. Kandel. Cognitive Neuroscience and the Study of Memory. *Neuron*, 20(3):445–468, March 1998.

[MT96] H. Markram and P. Tsodyks. Redistribution of Synaptic Efficacy Between Neocortical Pyramidal Neurons. *Nature*, 382(6594):807–810, August 1996.

[MWBC04] S. Modi, P.R. Wilson, A.D. Brown, and J. Chad. Behavioral simulation of biological neuron systems in SystemC. In *Proc. 2004 IEEE Int'l Behavioral Modeling and Simulation Conf.*, pages 31–36, 2004.

[NDKN07] J. M. Nageswaran, N. Dutt, J. L. Krichmar, and A. Nicolau. A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. *Neural Networks*, 22(5–6), July/August 2007.

[NdSMdS09] N. Nedjah, R. M. da Silva, L. M. Mourelle, and M. V. C. da Silva. Dynamic MAC-based architecture of artificial neural networks suitable for hardware implementation on FPGAs. *Neurocomputing*, 72(10–12):2171–2179, June 2009.

[NG03] B. Noory and V. Groza. A Reconfigurable Approach to Hardware Implementation of Neural Networks. In *Proc. 2003 Canadian Conf. on Electrical and Computer Engineering (CCECE 2003)*, pages 1861–1864, 2003.

[NPMA$^+$10] J. Navaridas, L. A. Plana, J. Miguel-Alonso, M. Luján, and S. B. Furber. SpiNNaker: Effects of Traffic Locality and Causality on the Performance of the Interconnection Network. In *Proc. 2010 ACM Int'l Conf. Computing Frontiers*, pages 11–19, 2010.

[NS92] T. Nordström and B. Svensson. Using and Designing Massively Parallel Computers for Artificial Neural Networks. *J. Parallel and Distributed Computing*, 14(3):260–285, March 1992.

[OCR$^+$06] E. M. Ortigosa, A. Cañas, E. Ros, P. M. Ortigosa, S. Mota, and J. Diaz. Hardware description of multi-layer perceptrons with different abstraction levels. *Microprocessors and Microsystems*, 30(7):435–444, November 2006.

[OJ04] K.-S. Oh and K. Jung. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, June 2004.

[Omo00] A. R. Omondi. Neurocomputers: a dead end? *Int'l J. Neural Systems*, 10(6), June 2000.

[OWLD05] M. Oster, A. M. Whatley, S.-C. Liu, and R. J. Douglas. A Hardware/Software Framework for Real-Time Spiking Systems. In *Proc. 15th Int'l Conf. Artificial Neural Networks (ICANN 2005)*, pages 161–166, 2005.

[PA05] M. S. Prieto and A. R. Allen. A hybrid system for embedded machine vision using FPGAs and neural networks. *Machine Vision and Applications*, 20(6):379–384, October 2005.

[Pan01] P.R. Panda. SystemC - a modeling platform supporting multiple design abstractions. In *Proc. Int'l Symp. on System Synthesis*, 2001.

[PEM+07] H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig. Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers. In *Proc. 13th Int'l Euro-Par Conf. on Parallel Processing (Euro-Par 2007)*, pages 672–681, 2007.

[PFT+07] L. A. Plana, S. B. Furber, S. Temple, M. M. Khan, Y. Shi, J. Wu, and S. Yang. A GALS Infrastructure for a Massively Parallel Multiprocessor. *IEEE Design & Test of Computers*, 24(5):454–463, Sep.-Oct. 2007.

[PGDK88] D. A. Pomerleau, G.L. Gusciora, D.S.Touretsky, and H.T. Kung. Neural Network Simulation at WARP Speed: How We Got 17 Million Connections Per Second. In *Proc. IEEE Int'l Conf on Neural Networks*, pages 143–150, 1988.

[PKP08] P. M. Papazoglou, D. A. Karras, and R. C. Papademetriou. On Improved Event Scheduling Mechanisms for Wireless Communications Simulation Modelling. In *Proc. New Technologies, Mobility and Security Conf. and Wkshps. (NTMS2008)*, pages 1–5, 2008.

[PLGW07] A. W. Przybyszewski, P. Linsay, P. Gaudiano, and C. M. Wilson. Basic Difference Between Brain and Computer: Integration of Asynchronous Processes Implemented as Hardware Model of the Retina. *IEEE Trans. Neural Networks*, 18(1):70–85, January 2007.

[PMP+05] M. J. Pearson, C. Melhuish, A. G. Pipe, M. Nibouche, I. Gilhespy, K. Gurney, and M. Mitchinson. Design and FPGA implementation of an embedded real-time biologically plausible spiking neural network processor. In *Proc. 2005 Int'l Conf. on Field-Programmable Logic and Applications (FPL'05)*, pages 5822–585, 2005.

[PPM$^+$07] M. J. Pearson, A. G. Pipe, B. Mitchinson, K. Gurney, C. Melhuish, I. Gilhespy, and M. Nibouche. Implementing Spiking Neural Networks for Real-Time Signal-Processing and Control Applications: A Model-Validated FPGA Approach. *IEEE Trans. Neural Networks*, 18(5):1472–1487, September 2007.

[Pra08] R. D. Prabhu. SOMGPU: An Unsupervised Pattern Classifier on Graphical Processing Unit. In *Proc. 2008 IEEE Cong. on Evolutionary Computation (CEC 2008)*, pages 1011–1018, 2008.

[PRAP97] F. J. Pelayo, E. Ros, X. Arreguit, and A. Prieto. VLSI Implementation of a Neural Model Using Spikes. *Analog Integrated Circuits and Signal Processing*, 13(1–2):111–121, May–June 1997.

[PUS96] E. Pérez-Uribe and E. Sanchez. FPGA Implementation of an Adaptable-Size Neural Network. In *Proc. 1996 Int'l Conf. Artificial Neural Networks (ICANN 96)*, pages 383–388, 1996.

[PVLBC$^+$06] R. Paz-Vicente, A. Linares-Barranco, D. Cascado, M. A. Rodriguez, G. Jimenez, A. Civit, and J. L. Sevillano. PCI-AER interface for Neuro-inspired Spiking [Systems. In *Proc. 2006 IEEE Int'l Symp. Circuits and Systems (ISCAS 2006)*, pages 3161–3164, 2006.

[PWKR02a] M. Porrmann, U. Witkowski, H. Kalte, and U. Rückert. Dynamically Reconfigurable Hardware - A New Perspective for Neural Network Implementations. In *Proc. 2002 Int'l Conf. Field Programmable Logic and Applications (FPL 2002)*, pages 1048–1057, 2002.

[PWKR02b] 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.

[RD07] M. Rudolph and A. Destexhe. How much can we trust neural simulation strategies? *Neurocomputing*, 70(10–12):1966–1969, June 2007.

[Rey03] L. M. Reyneri. Implementation Issues of Neuro-Fuzzy Hardware: Going Towards HW/SW Codesign. *IEEE Trans. Neural Networks*, 14(1):176–194, January 2003.

[RJG$^+$10] A. D. Rast, X. Jin, F. Galluppi, L. A. Plana, C. Patterson, and S. B. Furber. Scalable Event-Driven Native Parallel Processing: The SpiNNaker Neuromimetic System. In *Proc. 2010 ACM Conf. Computing Frontiers (CF'10)*, pages 20–29, 2010.

[RJKF09] A. Rast, X. Jin, M. Khan, and S. Furber. The deferred-event model for hardware-oriented spiking neural networks. In *Proc. 2008 Int'l Conf. Neural Information Processing (ICONIP 2008)*, pages 1057–1064. Springer-Verlag, 2009.

[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)*, pages 2611–2618, 2009.

[ROA$^+$06] E. Ros, E. M. Ortigosa, R. Agís, R. Carrillo, and M. Arnold. Real-Time Computing Platform for Spiking Neurons (RT-Spike). *IEEE Trans. Neural Networks*, 17(4):1050–1063, July 2006.

[Ros58] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psych. Review*, 65:286–408, 1958.

[RTB$^+$07] S. Renaud, J. Tomas, Y. Bornat, A. Daouzli, and S. Saïgli. Neuromimetic ICs With Analog Cores: An Alternative for Simulating Spiking Neural Networks. In *Proc. IEEE Int'l Symp. Circuits and Systems (ISCAS2007)*, pages 3355–3358, 2007.

[Rüc01] U. Rückert. ULSI Architectures for Artificial Neural Networks. In *Proc. 9th Euromicro Wkshp. on Parallel and Distributed Processing*, pages 436–442, 2001.

[RYKF08] A.D. Rast, S. Yang, M. M. Khan, and S.B. Furber. Virtual Synaptic Interconnect Using an Asynchronous Network-on-Chip. In *Proc. 2008 Int'l Joint Conf. Neural Networks (IJCNN2008)*, pages 2727–2734, 2008.

[SAMK02] T. Schoenauer, S. Atasoy, N. Mehrtash, and H. Klar. NeuroPipe-Chip: A Digital Neuro-Processor for Spiking Neural Networks. *IEEE Trans. Neural Networks*, 13(1):205–213, January 2002.

[SBC02] H. Z. Shouval, M. F. Bear, and L. N. Cooper. A unified model of NMDA receptor-dependent bidirectional synaptic plasticity. *Proc. Nat. Acad. of Sciences of the USA*, 99(16):10831–10836, August 2002.

[SBS05] D. Steinkraus, I. Buck, and P. Y. Simard. Using GPUs for machine learning algorithms. In *Proc. 8th Int'l Conf. Document Analysis and Recognition*, pages 1115–1120, 2005.

[SFM08] J. Schemmel, J. Fieres, and K. Meier. Wafer-Scale Integration of Analog Neural Networks. In *Proc. 2008 Int'l Joint Conf. Neural Networks (IJCNN2008)*, pages 431–438, 2008.

[SHMS04] J. Schemmel, S. Hohmann, K. Meier, and F. Schürmann. A Mixed-Mode Analog Neural Network Using Current-Steering Synapses. *Analog Integrated Circuits and Signal Processing*, 38(2–3):233–244, Jan–Feb 2004.

[SHS95] M. Scholles, B. J. Hosticka, and M. Schwarz. Real-Time Application of Biology-Inspired Neural Networks Using an Emulator with Dedicated Communication Hardware. In *Proc. 1995 IEEE Int'l Symp. Circuits and Systems (ISCAS1995)*, pages 267–270, 1995.

[SKIO95] T. Shibata, H. Kosaka, H. Ishii, and T. Ohmi. A Neuron-MOS Neural Network Using Self-Learning-Compatible Synapse Circuits. *IEEEjssc*, 30(8):913–922, August 1995.

[ŠO03] J. Šíma and P. Orponen. General-Purpose Computation with Neural Networks: A Survey of Complexity Theoretic Results. *Neural Computation*, 15(12):2727–2778, December 2003.

[ST01] O. Sporns and G. Tononi. Classes of Network Connectivity and Dynamics. *Complexity*, 7(1):28–38, sep–oct 2001.

[STE00] O. Sporns, G. Tononi, and G. M. Edelman. Connectivity and complexity: the relationship between neuroanatomy and brain dynamics. *Neural Networks*, 13(8):909–922, August 2000.

[Str97] B. Stroustrup. *The C++ Programming Language*, chapter 23, page 712. Addison-Wesley, Reading, Mass., 1997.

[SVS$^+$09] J. Strunk, T. Volkmer, K. Stephan, W. Rehm, and H. Schick. Impact of Run-Time Reconfiguration on Design and Speed - A Case Study Based on a Grid of Run-Time Reconfigurable Modules inside a FPGA. In *Proc. 2009 Int'l Symp. Parallel and Distributed Processing Systems (IPDPS 2009)*, pages 1048–1057, 2009.

[Tay03] J. G. Taylor. The CODAM model of Attention and Consciousness. In *Proc. 2003 Int'l Joint Conf. Neural Networks (IJCNN2003)*, pages 292–297, 2003.

[TBS$^+$06] J. Tomas, Y. Bornat, S. Saïgli, T. Lévi, and S. Renaud. Design of a modular and mixed neuromimetic ASIC. In *Proc. 13th IEEE Int'l Conf. Electronics, Circuits, and Systems (ICECS2006)*, pages 946–949, 2006.

[TH86] D. W. Tank and J. J. Hopfield. Simple "Neural" Optimization Networks: an A/D Converter, Signal Decision Circuit, and a Linear Programming Circuit. *IEEE Trans. Circuits and Systems*, 33(5):533–541, May 1986.

[THA$^+$10] I. T. Tokuda, C. E. Han, K. Aihara, M. Kawato, and N. Schweighofer. The role of chaotic resonance in cerebellar learning. *Neural Networks*, 23(7):836–842, September 2010.

[THGG07] C. Torres-Huitzil, B. Girau, and A. Gauffriau. Hardware/Software Codesign for Embedded Implementation of Neural Networks. In *Proc. Reconfigurable Computing: Architectures, Tools, and Applications, 3rd Int'l Wkshp. (ARC 2007)*, pages 167–178, 2007.

[TL09] D. B. Thomas and W. Luk. FPGA Accelerated Simulation of Biologically Plausible Neural Networks. In *Proc. 17th IEEE Symp. on Field Programmable Custom Computing Machines (FCCM2009)*, pages 45–52, 2009.

[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. Springer-Verlag, 2006.

[TMS93] M. Tsodyks, I. Mitkov, and H. Sompolinski. Patterns of Synchrony in Inhomogeneous Networks of Oscillators with Pulse Interactions. *Phys. Review Letters*, 71(8):76–78, August 1993.

[TS01] P. H. E Tiesenga and T. J. Sejnowski. Precision of pulse-coupled networks of integrate-and-fire neurons. *Network: Computation in Neural Systems*, 12(2):215–233, May 2001.

[UPRS05] A. Upegui, C. A. Peña-Reyes, and E. Sanchez. An FPGA platform for on-line topology exploration of spiking neural networks. *Microprocessors and Microsystems*, 29(5):211–223, June 2005.

[UVCP98] A. Uncini, L. Vecci, P. Campolucci, and F. Piazza. Complex-Valued Neural Networks with Adaptive Spline Activation Function for Digital Radio-Links Nonlinear Equalization. *IEEE Trans. Signal Processing*, 47(2):505–514, February 1998.

[VA05] T. P. Vogels and L. F. Abbott. Signal Propagation and Logic Gating in Networks of Integrate-and-Fire Neurons. *J. Neuroscience*, 25(46):10786–10795, November 2005.

[VMC$^+$07] R. J. Vogelstein, U. Mallik, E. Culurciello, G. Cauwenberghs, and R. Etienne-Cummings. A Multichip Neuromorphic System for Spike-Based Visual Information Processing. *Neural Computation*, 19(9):2281–2300, September 2007.

[VMVC07] R. J. Vogelstein, U. Mallik, J. T. Vogelstein, and G. Cauwenberghs. Dynamically Reconfigurable Silicon Array of Spiking Neurons With Conductance-Based Synapses. *IEEE Trans. Neural Networks*, 18(1):253–265, January 2007.

[WCIS07] Hsi-Ping Wang, E. Chicca, G. Indiveri, and T. J. Sejnowski. Reliable Computation in Noisy Backgrounds Using Real-Time Neuromorphic Hardware. In *Proc. 2007 IEEE Biomedical Circuits and Systems Conf. (BIOCAS2007)*, pages 71–74, 2007.

[WD08] J. H. B. Wijekoon and P. Dudek. Integrated Circuit Implementation of a Cortical Neuron. In *Proc. 2008 IEEE Int'l Symp. Circuits and Systems (ISCAS2008)*, pages 1784–1787, 2008.

[WF09] J. Wu and S. B. Furber. A Multicast Routing Scheme for a Universal Spiking Neural Network Architecture. *Computer Journal*, 53(3):280–288, March 2009.

[WL06] R. K. Weinstein and R. H. Lee. Architectures for high-performance FPGA implementations of neural models. *J. Neural Engineering*, 3(1):21–34, March 2006.

[WNE97] W. C. Westerman, D. P. M. Northmore, and J. G. Elias. Neuromorphic Synapses for Artificial Dendrites. *Analog Integrated Circuits and Signal Processing*, 13(1–2):167–184, May–June 1997.

[WNE99] W. C. Westerman, D. P. M. Northmore, and J. G. Elias. Antidromic Spikes Drive Hebbian Learning in an Artificial Dendritic Tree. *Analog Integrated Circuits and Signal Processing*, 18(2–3):141–152, February 1999.

[WR07] S. Welbourne and M. A. Lambon Ralph. Using parallel distributed processing models to simulate phonological dyslexia:

The key role of plasticity related recovery. *J. Cognitive Neuroscience*, 19(7), July 2007.

[YB09] Z. Yu and B. M. Baas. High Performance, Energy Efficiency, and Scalability With GALS Chip Multiprocessors. *IEEE Trans. VLSI Systems*, 17(1):66–79, January 2009.

[YC09] T. Yu and G. Cauwenberghs. Analog VLSI Neuromorphic Network with Programmable Membrane Channel Kinetics. In *Proc. 2009 IEEE Int'l Symp. Circuits and Systems (ISCAS 2009)*, pages 349–352, 2009.

[YC10] T. Yu and G. Cauwenberghs. Analog VLSI Biophysical Neurons and Synapses With Programmable Membrane Channel Kinetics. *IEEE Trans. Biomedical Circuits and Systems*, 4(3):139–148, June 2010.

[YMY$^+$90] M. Yasunaga, N. Masuda, M. Yagyu, M. Asai, M. Yamada, and A. Masaki. Design, Fabrication and Evaluation of a 5-inch Wafer Scale Neural Network LSI Composed of 576 Digital Neurons. In *Proc. 1990 Int'l Joint Conf. Neural Networks (IJCNN1990)*, pages 527–535, 1990.

[ZB06] K. A. Zaghloul and K. Boahen. A silicon retina that reproduces signals in the optic nerve. *J. Neural Engineering*, 3(4):257–267, December 2006.