International Journal of Parallel Programming manuscript No.
(will be inserted by the editor)

Managing Burstiness and Scalability in Event-Driven Models on the SpiNNaker Neuromimetic System

Alexander D. Rast · Javier Navaridas · Xin Jin · Francesco Galluppi · Luis A. Plana · Jose Miguel-Alonso · Cameron Patterson · Mikel Luján · Steve Furber ·

Received: date / Accepted: date

Abstract Neural networks present a fundamentally different model of computation from the conventional sequential digital model, for which conventional hardware is typically poorly matched. However, a combination of model and scalability limitations has meant that neither dedicated neural chips nor FPGA's have offered an entirely satisfactory solution. SpiNNaker introduces a different approach, the "neuromimetic" architecture, that maintains the neural optimisation of dedicated chips while offering FPGA-like universal configurability. This parallel multiprocessor employs an asynchronous event-driven model that uses interrupt-generating dedicated hardware on the chip to support real-time neural simulation. Nonetheless, event handling, particularly packet servicing, requires careful and innovative design in order to avoid local processor congestion and possible deadlock. We explore the impact that spatial locality, temporal causality and burstiness of traffic have on network performance, using tunable, biologically similar synthetic traffic patterns. Having established the viability of the system for real-time operation, we use two exemplar neural models to illustrate how to implement efficient event-handling service routines that mitigate the problem of burstiness in the traffic. Extending work published in ACM Computing Frontiers 2010 with on-chip testing, simulation results indicate the viability of SpiNNaker for large-scale neural modelling, while emphasizing the need for effective burst management and network mapping. Ultimately, the goal is the creation of a library-based development system that can translate a high-level neural model from any description environment into an efficient SpiNNaker instantiation. The complete system represents a general-purpose platform that can generate an arbitrary neural network and run it with hardware speed and scale.

Alexander Rast
School of Computer Science
University of Manchester
Oxford Road
Manchester, UK M13 9PL
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: rasta@cs.man.ac.uk

Keywords Asynchronous · Burst · Network · Event-driven · Universal · Neural · Multiprocessor · Interconnection · Real-time · Traffic · Characterisation

CR Subject Classification C.1.3

1 Introduction

Neural networks present an emphatically different model of computation from the conventional sequential digital model. This make it unclear, at best, whether running neural networks on industry-standard computer architectures represents a good, much less an optimum, implementation strategy. Such concerns have become particularly pressing with the emergence of large-scale spiking models [22] attempting biologically realistic simulation of brain-scale networks. While dedicated hardware is thus becoming increasingly attractive [28], it is also becoming clear that a fixed-model design would be a poor choice [55], given that just as there is debate over the architectural model in the computational community, there is no consensus on the correct model of the neuron in the biological community [23]. Our proposed solution is the "neuromimetic" architecture: a system whose hardware retains enough of the native parallelism and asynchronous event-driven dynamics of "real" neural systems to be an analogue of the brain, enough general-purpose programmability to experiment with arbitrary biological and computational models. This neuromimetic device, SpiNNaker, is a scalable universal neural network chip that for the first time provides a hardware platform for neural model exploration able to support large-scale networks with millions of neurons.

The SpiNNaker chip (fig. 1) is a plastic platform containing configurable blocks of generic processing and connectivity whose structure and function are designed and optimised for neural computation. This distinguishes it strongly from completely general-purpose FPGA's and also from dedicated devices that offer a fixed selection of neural models. The primary features of the neuromimetic architecture are:

Native Parallelism: There are multiple processors per device, each operating completely independently from each other.

Event-Driven Processing: An external, self-contained, instantaneous signal drives state change in each process, which contains a trigger that will initiate or alter the process flow,

Incoherent Memory: Any processor may modify any memory location it can access without notifying or synchronising with other processors.

Incremental Reconfiguration: The structural configuration of the hardware can change dynamically while the system is running.

These characteristics mean SpiNNaker has an entirely different model of computation from the conventional sequential one. In ACM Computing Frontiers 2010, we used SpiNNaker simulations as an example to illustrate the differences between asynchronous and conventional parallel processing [51], [41]. Here, we demonstrate the example in practice with additional models tested on the physical hardware.

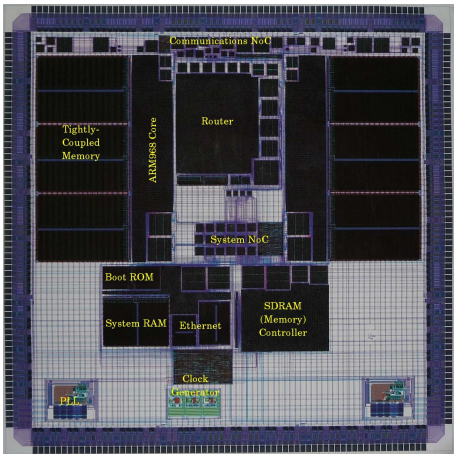


Fig. 1 SpiNNaker test chip.

2 Neural System Architectures

Neural networks are parallel processing architectures that involve simple atomic computations occurring in individual elements - neurons - interconnected among each other through links - synapses - that themselves perform some limited computation. In addition, synapses modify their computation in time (over timescales long compared to the process dynamics of neurons), usually by adjusting a weight which simply scales the relative contribution of the individual synapse. Many modern neural networks use spiking dynamics, involving a solution of differential equations for each neuron's state: these are typical of biological models. Some networks, aimed at purely computational applications, forego this continuous-time differential formulation in favour of a discrete-time process that simply evaluates a static nonlinearity at the neuron. Regardless of the model used, however, the question of the appropriate computational platform for a neural network simulation has been one of the dominant topics in the field.

2.1 Pure Software Simulation

The conventional way, and still by far the most widely-used method, to simulate neural networks is through software simulation on conventional computers. The computing platform may vary all the way from a single uniprocessor PC [21], through PC clusters [37] [42], to large mainframes [36] [2]. Software is equally varied but tends to depend strongly on the research domain. For biologically realistic modelling at the microscopic level with fully accurate dynamics, the dominant applications are NEURON [18] and GENESIS (http://genesis-sim.org). Simulators like Brian [13] are in common use for dynamic-level simulation where complete biological realism is secondary to the basic dynamics at the spiking level. Such software tends to abstract

neurons to a spatial point, and spikes to zero-time events. In the realm of artificial neural networks for computing applications, software such as JNNS (http://www-ra.informatik.uni-tuebingen.de/software/JavaNNS/welcome_e.html) has seen some use, although these applications are waning with the emergence of spiking networks. Finally, many users use Matlab [23] or C/C++ [52] to write their own neural simulators.

Software simulation tends to be slow and may require large computers for detailed simulations on large-scale models. To improve performance, recent software tools have turned to event-driven computing [58] [34] [7]. However, conventional sequential computers do not usually have direct hardware support for event-driven applications, and thus most event-driven simulators actually run an emulation by using a small timestep, recording events in an event queue, and updating all processes dependent upon the events in the queue at the appropriate timestep [32], [53]. While this improves efficiency over fully synchronous approaches, it still encounters limitations with very large networks that require either using simple dynamics such as leaky integrate-and-fire, or modelling populations of neurons as a single object rather than each individual neuron.

2.2 Adapted General-Purpose Hardware

The emergence of various general-purpose devices supporting some level of parallel processing has generated numerous attempts to map various neural algorithms to the hardware. A remarkable early attempt using a processor with strong similarities to SpiNNaker, the Datawave chip [24], appears not to have been pursued further because of the limited commercial success and eventual disappearance of the hardware. While the increasing ubiquity of standard multicore microprocessors introduces an obvious opportunity to exploit parallelism, other, more creative approaches use field-programmable gate arrays (FPGA's) [45] and graphics processor units (GPU's) [38]. FPGA's, in particular, offer an attractive possibility: reconfigurable computing. In reconfigurable architectures, the model can modify the hardware configuration of the chip while the simulation is running. Either through component swapping [12] or network remapping [56], these approaches seek to circumvent scalability limitations, with some success, but with both FPGA's and GPU's scalability has proven to be the main problem, with FPGA's running into routing barriers due to their circuit-switched fabric [33] and GPU's running into memory access barriers. Even more problematic has been power consumption: a typical large FPGA may dissipate ~ 50W and a GPU accelerator ~ 200W. Thus adapting general-purpose hardware seems to be a realistic approach only for small-scale model prototyping.

2.3 Dedicated Neural Hardware

Given the limitations of off-the-shelf hardware, many groups have implemented dedicated neural hardware systems, usually involving a custom IC. Attempts began as early as the late 1980's [14], [9]. This approach yields the greatest scope for architectural diversity as well as performance: different designs have used analogue [19] or

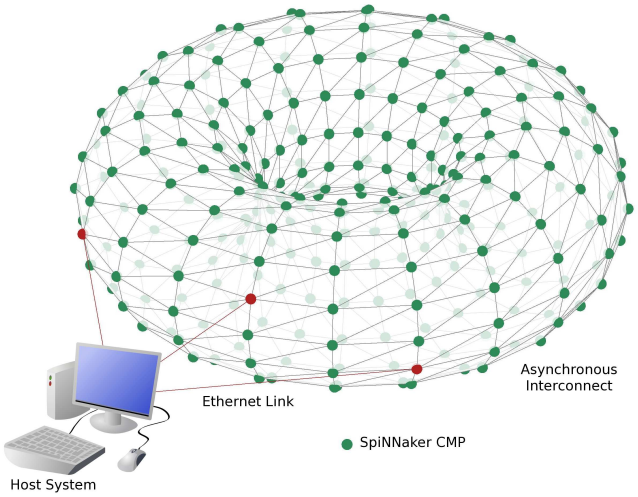


Fig. 2 SpiNNaker system topology.

digital [59] technology, hardwired [4] or configurable [57] architecture, continuous-activation [31] or spiking [43] signalling, coarse- [54] or fine-grained [8] parallelism. In recent years, however, interest has moved primarily towards processors for the simulation of spiking neural networks. Here again there have been two threads of development. In the "neuromorphic" approach [20], chips use analogue circuitry to emulate as closely as possible the actual biophysics of real neurons [60]. The "neuroprocessor" approach [35], by contrast, attempts to use general-purpose digital components with an internal structure optimised for massively parallel neural processing. Each has its limitations: neuromorphic chips are power- and component-efficient, but relatively small-scale, and have limited or fixed model support. Neuroprocessors have, to date, suffered from interconnect limitations, a combination of limited bus bandwidth, synchronous shared-access protocols, and circuit-switched architecture [16]. Thus, despite the obvious speed improvements, dedicated neural devices have not thus far achieved the scalability that would permit truly large-scale simulation, due to hardware limitations. To minimise such limitations while providing the neural acceleration that only dedicated hardware can provide, we have introduced the SpiNNaker neuromimetic architecture.

3 The SpiNNaker Neuromimetic IC

The SpiNNaker chip is the core building block component of a large-scale system using an array of chips arranged in a 2-dimensional triangular torus topology (fig. 2). Using this diagonal-link topology increases system robustness through the connection redundancy inherent in the toroidal physical topology, while permitting an arbitrary mapping of large-scale neural networks to physical chips and links.

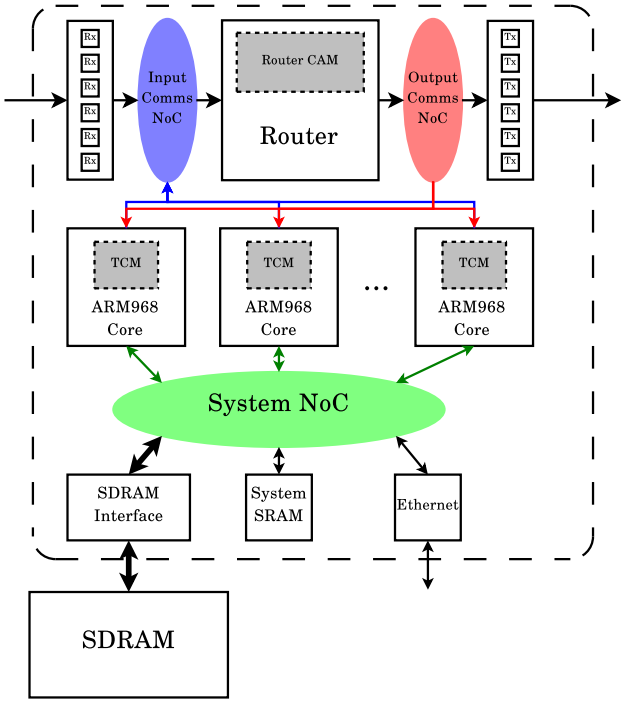


Fig. 3 SpiNNaker Architecture. The dashed box indicates the extent of the SpiNNaker chip. Dotted grey boxes indicate local memory areas.

3.1 Implementation of the 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 as practicable explicitly in hardware for maximal performance [51]. It implements these key architectural features using a mixture of off-the-shelf and custom components. By design the system is optimised for spiking models, but this does not constrain it exclusively to spiking neural networks. We identify four features as fundamental to the neuromimetic architecture.

3.1.1 Native Parallelism

"Real" neural networks are massively parallel processors. Native parallelism is therefore basic to the neuromimetic architecture. SpiNNaker (fig. 3) contains multiple (2 in the test implementation, 18 in a recently fabricated version) independent ARM968 processors, each simulating a variable number of neurons which could be as few as 1 or as many as 1,700. Each processor operates entirely independently (on separate clocks) and has its own private subsystem containing various devices to support neural functionality. The principal devices are a communications controller that handles

input and output traffic in the form of “spike” packets, a DMA controller that provides fast virtual access to synaptic data residing off-chip in a separate memory, and a Timer that supports the generation of fixed time steps where models need them. The entire subsystem is therefore a self-contained processing element modelling a neural group. This “processing node” is truly concurrent, in that it uses only local information to control execution and operates asynchronously from other processing nodes.

3.1.2 Event-Driven Processing

Biological neurons communicate primarily through spikes: short-duration impulses whose precise shape is usually considered immaterial. Spikes appear to function as events - essentially point processes. SpiNNaker’s communication network is a configurable packet-switched asynchronous interconnect using Address-Event Representation (AER) [30] to transmit neural signals between processors. AER is an emerging neural communication standard [3] that abstracts spikes from neurobiology into a single atomic event, transmitting only the address of the neuron that fired; SpiNNaker extends this basic standard with an optional 32-bit payload. The interconnect itself extends both on-chip and off-chip as the Communications Network-on-Chip (Comms NoC). Previous work ([44], [29]) describes the design of and configuration procedure for the Comms NoC. At the processor node, the communications controller receives and generates AER spikes, issuing an interrupt (i.e., an event) to the processor when a new packet arrives. From the point of view of the neuromimetic architecture, this fabric implements the support infrastructure for incremental reconfiguration and the event-driven model.

3.1.3 Incoherent Distributed Memory

The notion of controlled shared access to a central memory store simply does not exist in biology; neurons update using purely local information. Thus any processor may modify any memory location it can access without notifying or synchronising with other processors. SpiNNaker processors have access to 2 primary memory resources: their own local “Tightly-Coupled Memory” (TCM) and a global SDRAM device, neither of which require or have support for coherence mechanisms. The TCM is only accessible to its own processor and contains both the executing code (in the “Instruction TCM” (ITCM)) and any variables that must be accessible on-demand (in the “Data TCM” (DTCM)). The global SDRAM contains the synaptic data (and possibly other large data structures). Since each synapse in the SDRAM connects to a single target neuron residing in a specific processor, the SDRAM is segmented into discrete regions for each processor, grouped by postsynaptic neuron. This obviates the need for coherence checking because only one processor node will access a given address range. At the processor node, the DMA controller makes the synapse appear virtually local to the processor by bringing its data into DTCM when an incoming packet arrives [49]. The DMA controller also generates an event - DMA complete - when the entire synaptic block has been transferred into local memory. Overall

therefore, the SDRAM behaves more as an extension of local memory into a large off-chip area than a shared memory area, and thus from a system point of view, effectively all memory is local.

3.1.4 Incremental Reconfiguration

Biological neural networks are plastic: the physical topology changes during operation. Likewise, the structural configuration of neuromimetic hardware can change dynamically while the system is running. SpiNNaker uses a distributed routing subsystem to direct packets through the Comms NoC, which converts spike events into AER packets. Each chip has a packet-switching router that handles these packets and distributes them seamlessly to all connected neurons through the GALS interconnect. The design of the router incorporates a multicast diffusion mechanism devised to support biologically realistic neural fan-out (~ 1000 connections/neuron) A 1024-word associative routing table within each router defines the neural connectivity. To minimise the risk of local failure, an "emergency routing" mechanism allows bypass of a failed link using an automated routing algorithm that routes packets in a triangular path around a local link obstruction. Routes are fully reprogrammable by changing the routing table, just as the model dynamics are reprogrammable by swapping the running code, making it possible, at least in principle, to reconfigure the model on the fly.

3.2 Nondeterministic process dynamics

While this event-driven solution is far more scalable than either synchronous or circuit-switched systems, it presents significant implementation challenges when the network is large and packet traffic dense.

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 starvation, 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. The router includes a *limited* Emergency Routing mechanism that

avoids congested or malfunctioning links by routing packets through the next clockwise port. 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, and which demonstrates much about the concurrent model of computation.

4 Event-Driven Processing

The software model uses a hardware-design-like flow based on hierarchical levels of abstraction. In a previous work [47] we introduced this 3-level software model for SpiNNaker, with a Model Level, a System Level, and a Device Level (fig. 4). The model defines an instantiation chain that proceeds from a behavioural neural model down to a specific machine-level implementation.

4.1 The event-driven model at the Model Level

Model Level treats the system as a process abstraction that hides all the hardware detail and considers the model purely in terms of neural properties. 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. New input spikes trigger update of the dynamics. In nonspiking networks different abstractions are necessary. One easy and common method is time sampling: events could happen at a fixed time interval, and this periodic event signal triggers the update. Alternatively, to reduce event rate with slowly-variable signals, a neuron may only generate 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. Decisions about the event representation at the Model Level could be almost entirely arbitrary, but in order to implement the model efficiently on SpiNNaker the representation chosen should have a simple correspondence to the physical hardware. Therefore, Model Level does not define the event representation, but rather has an interface to automated tools that generate the mapping operating at a lower level, one which has visibility both of SpiNNaker hardware and of the Model-Level definitions.

4.2 The event-driven model at the System Level

System Level is the level that provides visibility both of the model and of SpiNNaker. At the system level the internal components of SpiNNaker become visible, but only as

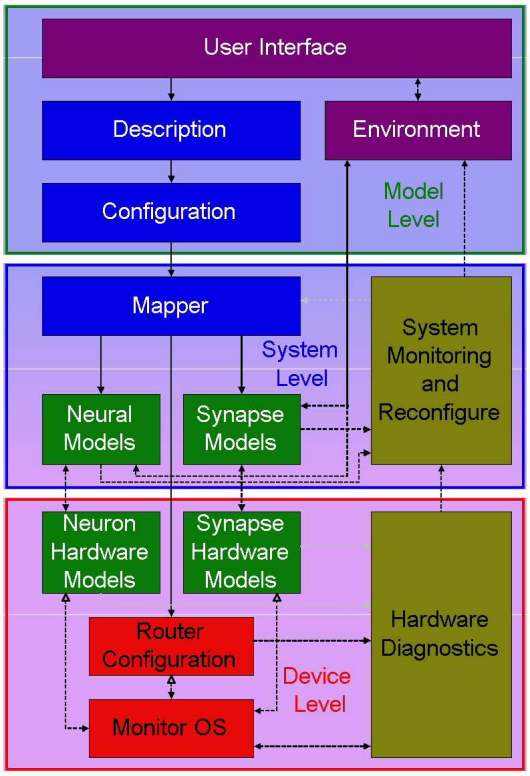


Fig. 4 SpiNNaker Software Model

high-level objects. At this level, events are transactions between objects representing individual components. Responses to events are the subroutine calls (or methods) to execute when the event arrives. These callbacks will be different for different neural models, and because automated tools must be able to associate a given model with a given series of SpiNNaker system objects, System Level is mostly a collection of libraries for different neural models. Each library defines the event representation as set of callback functions: a Packet-Received event, a DMA event, a Timer event, and an Internal (processor) event. It must also account for important system properties: no global state information and one-way communication. System-Level event functions must as a result use only local information, and if the current local information is insufficient to process the event, they must be able to transform it into a *future* event. There are several ways to do this: issue a DMA request, set a timer, or trigger an internal event. Given that much of the low-level hardware operation is common across all models, the System Level uses common device-driver support functions where possible, drawn from a base library written at a lower level.

4.3 The event-driven model at the Device Level

Device Level ignores the neural model altogether and considers SpiNNaker at the signalling level of its devices. At this level an event is its actual hardware nature: an interrupt, and the response likewise is the interrupt service routine (ISR) together with any deferred processes the ISR triggers. The hardware packet encoding is visible along with the physical registers in the DMA and communications controllers. Most of the Device Level code is therefore a series of interrupt-driven device drivers acting as support functions for the system level. Since Device Level code does not consider the neural model, these drivers are common across many models (and libraries), and includes operating-system-like system support, startup and configuration routines essential for the operation of the chip as a whole, but irrelevant from the point of view of the model. Device-Level ISR's must consider carefully asynchronous timing effects and the absence of network buffering: if the system expects a high event rate it needs to provide an event queue. As with any ISR, the objective is to defer as much processing as possible and exit the interrupt exception mode. Usually the deferred process is a System-Level function, so that the typical flow of control is that the System Level passes control to the Device-Level ISR when the initial event occurs, which then does the minimal processing necessary to capture the event and set/reset devices, then passes control back to the System-Level function. How this works in detail is easiest to see by considering actual model implementations on SpiNNaker.

5 Model Implementations

To test SpiNNaker functionality and performance, we have implemented both an abstract high-level model of the network and 3 different execution-level neural network models: 2 spiking models and a classical MLP model. The abstract model is designed to be a reasonably realistic approximation of network behaviour under typical operating conditions. We implemented the high level model in INSEE, a fast, flexible and mature simulation environment [40] for interconnection networks. The execution-level models are sufficiently different in design to form an effective first test of SpiNNaker universality while sufficiently representative to be reference examples for future model implementations. We first tested these models using ARM SoC Designer simulator, with additional low-level Verilog testing using Synopsys VCS, then ran the models on the SpiNNaker test chip in various network configurations.

5.1 Network system models

The developed network model contains most of the features of the router, as well as the topological arrangement. This study evaluates the largest possible system configuration: 64K nodes arranged on a 256×256 layout. The model of the router includes the emergency routing and deadlock avoidance mechanisms, whose parameters we set to the values suggested in [39]. To avoid coupling the evaluation to any particular biological network, table-based routing is not used (which significantly reduces

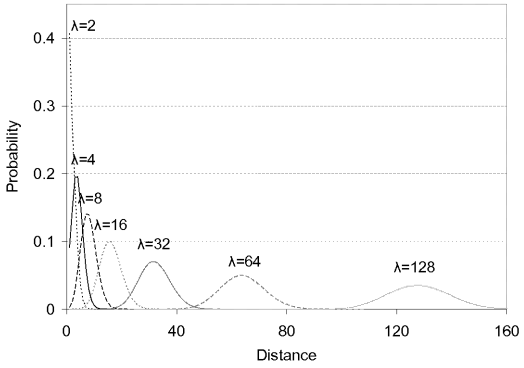


Fig. 5 Poisson distributions modelling traffic locality

the computing resources required to perform simulations). As actual routes between chips in the real system will attempt to use a minimal path with a single inflection point, the simulation sends packet through minimal routes using Dimension Order Routing (DOR) [5]. In DOR, diagonal links are considered to be a third dimension (Z). Routes followed by packets are always XY, XZ or YZ (an XYZ route cannot be a minimal path). The system models nodes as independent traffic sources that inject packets following a Bernoulli temporal distribution, with tunable packet injection rate i_r (packets/cycle/node). We provide them with the capability to react to receipt of a packet by generating a new packet or a burst of packets. Two parameters model such reactive traffic, the probability p to trigger a new packet (modelling causality) and the number of packets n that are triggered (simulating burstiness). The actual per-node generation rate G depends on both independent and reactive traffic:

$$G = i_r + i_r \cdot \sum_{k=1}^{\infty} n \cdot p^k$$

Models of the connection-level activity of brain-scale neural networks have thus far only described the general characteristics [15]. To simulate potentially realistic networks, we therefore use Poisson distributions that allow modelling of different degrees of *locality* by varying the lambda (λ) parameter. In general, the larger the value of λ the more distant the generated traffic. More specifically, we use seven different values from very local ($\lambda = 2$) to very far-flung ($\lambda = 128$). The traffic generation process is as follows. A sending node randomly selects a distance, d, following the given Poisson distribution, and then randomly selects a destination node, n, located d hops away. The node then injects a packet addressed to n. For the sake of simplicity, we restrict the study to a single distribution and use unicast rather than multicast packets. Figure 5 shows the distance distributions for each value of λ.

5.2 Spiking neural network models

Two models implement spiking neural networks, using either Izhikevich or Leaky-Integrate-and-Fire (LIF) neurons and Spike-Timing Dependent Plasticity (STDP) or α-amino-3-hydroxy-5-methyl-4-isoxazolepropionic acid (AMPA) synapses. The Izhikevich model [21] has been the reference spiking model driving design choices during hardware development because it is simple yet exhibits the full range of observed neural behaviour. We describe many of the algorithmic details of these models in the following papers: Izhikevich model [26], LIF model [50] STDP implementation [46], [27]. Here we focus on the event processing.

There are 3 main processes in the model, corresponding to the 3 event sources. The first process operates upon receipt of the input packet event, an interrupt from the communications controller. The second process operates upon receipt of the DMA completed event. The final process operates upon receipt of the Timer event.

5.2.1 Packet Received

The Packet Received event is a high-priority FIQ interrupt, in keeping with the use-it-or-lose-it nature: new packets must receive immediate, pre-emptive servicing or they will be lost. The process operates at Device Level and schedules the neural processing as a System Level function using the deferred-event model. When an input arrives, the process performs an associative lookup on the packet address to find a source ID. It then signals the DMA controller to transfer the corresponding row of synapses in memory into the next-available of an array of synaptic buffers, incrementing the available buffer number. It then exits and returns control to the scheduler.

5.2.2 DMA Completed

The DMA Completed, and all other events, are normal IRQ's, thus they may need to account for the arrival of other packets. In the case of DMA Completed, this means that its processing may not have completed before another DMA Completed event arrives, triggered by a packet arriving. The Device Level service routine therefore acknowledges the interrupt immediately, freeing the DMA controller for further transfers. Next, it tests the values of the synaptic buffer head and tail, to determine whether servicing of a DMA was still in progress when the next such interrupt arrived. If the difference is zero, i.e. no DMA service was in progress, it triggers a System Level deferred service process operating in User mode; otherwise it can simply return to the interrupted process. In its deferred service, the process goes through the synaptic buffers in sequence. For each buffer, it first performs synaptic weight updates, then computes the new contribution to the net input current at the delay value appropriate to each active synapse. Once it finishes with any given buffer, it updates the buffer queue head position, and if there remain buffers to service it continues on to the next one, exiting otherwise.

5.2.3 Timer

The Timer event has a higher priority than DMA Completed, since it operates on the current time rather than on future (delayed) time. Unlike the DMA Completed event no additional timer events can happen so it can operate continuously in an exception mode, however, to permit additional DMA interrupts it must exit from IRQ mode as soon as possible. Therefore, the Device Level service simply stacks registers and return addresses to the SVC (supervisor mode) stack, acknowledges the interrupt, and changes to supervisor mode. Operating at System Level in supervisor mode, it can avoid interfering with any potential deferred DMA operations still in progress while freeing the interrupt for DMA use. The SVC-mode process performs an efficient update of the neural states, computing and triggering any possible output spike, and with it update of postsynaptic information. The SVC-mode process must complete before the next (1ms) Timer interrupt.

5.3 MLP model

The second network is a classical multilayer perceptron (MLP) model using delta-rule backpropagation synapses with sigmoidal threshold neurons. The MLP is a broadly-used model ideal as a standard reference to test SpiNNaker's performance with non-spiking models. Some details of the model are in [48], however, this work largely discusses the topology and mapping. Here we consider the dynamics, or more accurately, the transformation of the MLP to a dynamic event-driven model. Since signals are continuous-valued "timeless" vectors, it is necessary to define an event representation for the dataflow. From this the process model will follow.

Because the mapping of the MLP to SpiNNaker (fig. 6) distributes unit processing among several processors (see [48] [25]) actual processing varies depending on whether the processor in concern implements the weight, sum, or threshold part of a unit.

5.3.1 MLP event representation

Representing MLP dynamics as events has two parts, a packet format and an event definition. 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 unit ID [1] and uses the payload to transmit that unit's activation. MLP neurons propagate input vectors between units in a unidirectional, feedforward manner: for each input presented one signal will pass over any given connection. Therefore one 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. This suggests another event: reverse-direction, that an individual unit can readily detect by triggering a software (SWI) event on Output Sent.

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

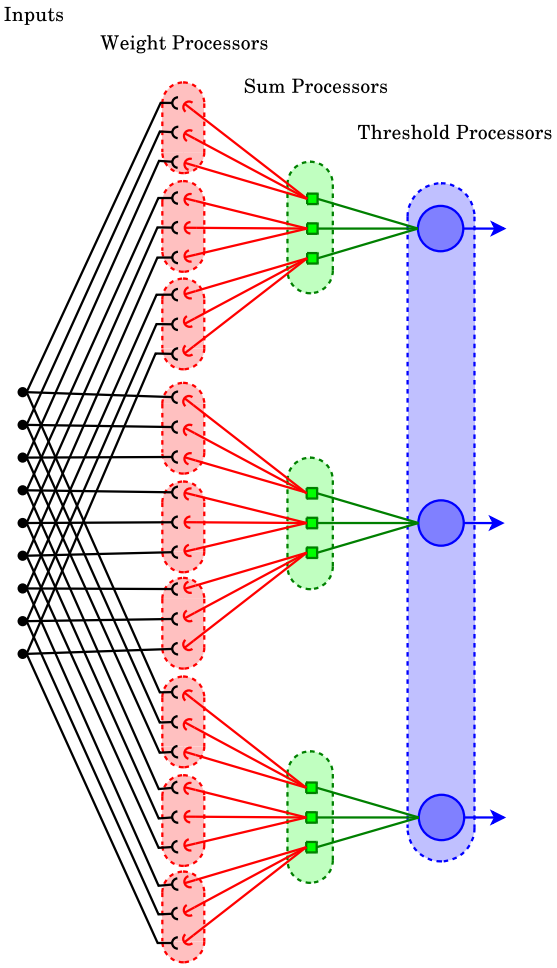


Fig. 6 SpiNNaker MLP mapping. Each dotted oval is one processor. Processors of all 3 types map a group of inputs i to outputs j. Each output j corresponds to a single neuron (or unit), while the inputs i depend on the stage. At each stage the unit sums the contributions from each previous stage. Note 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). It is possible to cascade sum processors to create a neuron with arbitrary fan-in.

These 2 events, importantly, 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.

5.3.2 Packet Received

The Packet Received event drives most of the MLP processing. Unlike the spiking case, the payload is critical, so the Device-level ISR for this event immediately places it in a queue for further processing. The rest of the processing occurs at System level.

Exact processing depends upon the stage; we denote the internal input variable as I and the output variable as J. The processing then goes as follows:

1. Dequeue a packet and 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 w_{ij} is the weight, and 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. (We have used standard momentum descent)
   (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 a spline-based interpolation to improve precision of J.
   (c) For threshold processors in the backward pass only, multiply the derivative by the new input: $J = J^b(LUT'(J^f))$.
   (d) Output a new packet with payload J.
4. If the packet queue is not empty, return to the start of the loop.

The critical concern with this process, since time is not a factor, is load balancing. If the processing in a given unit is much faster than other units, the packet traffic to the subsequent unit may become very bursty. This causes transient congestion, and *in extremis*, may deadlock the model. Obviously the sum processors have a trivial computation relative to weight and threshold, creating the potential for exactly such a problem. We developed 2 solutions: reduce the number of sums in any given stage (which for large fan-ins is the same as lengthening the processing pipeline), and combine the sum process with other processes, forcing it to compete for CPU time.

5.3.3 Output Sent

Output Sent occurs when all inputs have arrived, and the communications controller transitions to empty. At this point the processor triggers a FlipDirection SWI process that toggles the mode between forward and backward. The most general way to detecting the condition "all inputs arrived" 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. This could be a potential problem if neighbouring units had already sent while the current one still expected input. "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.

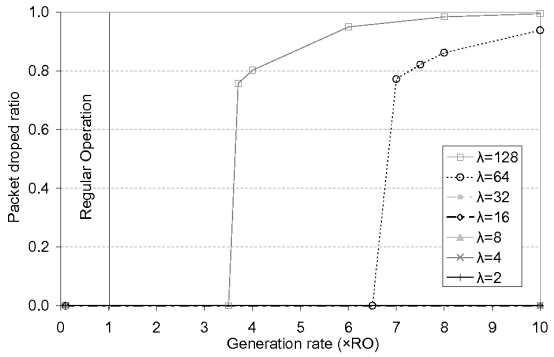


Fig. 7 Packet dropped ratio per configuration using Poisson-spatial traffic from independent sources. The X-axis shows the traffic generation rate and the Y-axis the measured ratio of dropped/injected packets.

6 Experimental Results

We ran simulations with 3 major objectives: packet processing performance, chip verification, and confirmation of accurate heterogeneous model support. Packet processing tests used INSEE [40] to assess the adequacy of the interconnection network. Simulations for chip verification used ARM SoC Designer Simulator on a complete SystemC model of the SpiNNaker chip. The heterogeneous model tests ran on the physical SpiNNaker test chip, using PyNN [6] as a standard simulation front-end. Our simulations used a 4-chip system with 2 processors per chip (corresponding to the first test board).

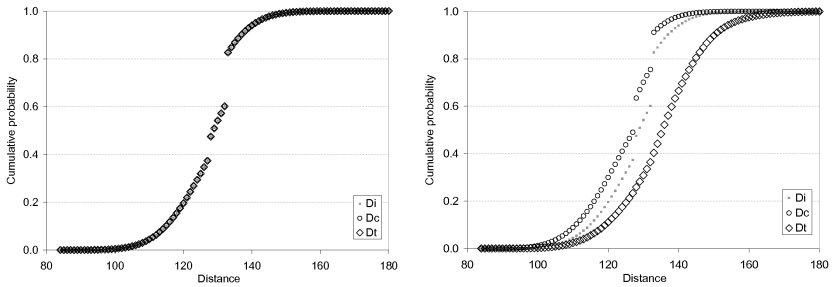
6.1 Packet Performance Testing

The first set of experiments focuses on measuring the impact that traffic locality, causality and burstiness have on the performance of the interconnection network. We modeled the workloads so that the locality, causality and burstiness of the traffic can be tuned using a collection of parameters. By sweeping over these parameters, we were able to assess the viability of the network interconnect in handling typical workloads during “Regular Operation” (RO) - a figure of merit derived in a previous work [39] from the maximum expected firing rate of neurons, the number of neurons supported in each chip and the packet size.

The locality experiments plotted in Fig. 7 used independent traffic sources (non-causal traffic) with various traffic generation rates from $0.1\times$RO (0.001 pkt/cyc/node) to $10\times$RO (0.1 pkt/cyc/node), and permit the observation of the relation between the degree of locality and the ratio of dropped packets. Most values are equal to zero: no packet is dropped, implying that the system behaves properly. Most configurations

can fully handle traffic with loads over 10 times those expected in the real system. Only those configurations where the traffic is sent to very distant areas ($\lambda = 128$ and $\lambda = 64$) show degradation, and even then it is limited to 3.7 and 7 times the maximum expected during regular operation of the system. This behaviour reinforces the impression of robustness encountered in previous experimentation [39]. At any rate, the load that the system can handle increases inversely with the distance. While lack of locality severely affects performance, it occurs at loads *well above* those required during regular operation of SpiNNaker. Because the system virtualises the network topology, an astute choice of mapping will almost always make it possible to keep routes mostly local, and thus degradation of network performance for load conditions in any event far outside the expected operating regime is of minimal concern.

Another interesting finding from the experiments is that, in those cases in which the network reaches saturation, the distance distribution computed at injection and that measured at consumption (considering those packets that are actually consumed) are noticeably different. Figure 8 shows the cumulative distance distribution of the system when being fed by the most distant traffic ($\lambda = 128$), at loads below and over the saturation point. Three figures of merit are plotted: the first one is the distance distribution at injection (D_i), computed as the number of hops in the shortest path between source and destination. The second is the distance distribution at consumption (D_c), also computed as the shortest path. Finally, the third is the distribution of the distance actually travelled by the packets, measured as the actual number of hops the packet traveled (D_t). Note that utilization of the emergency routing mechanism only affects D_t.

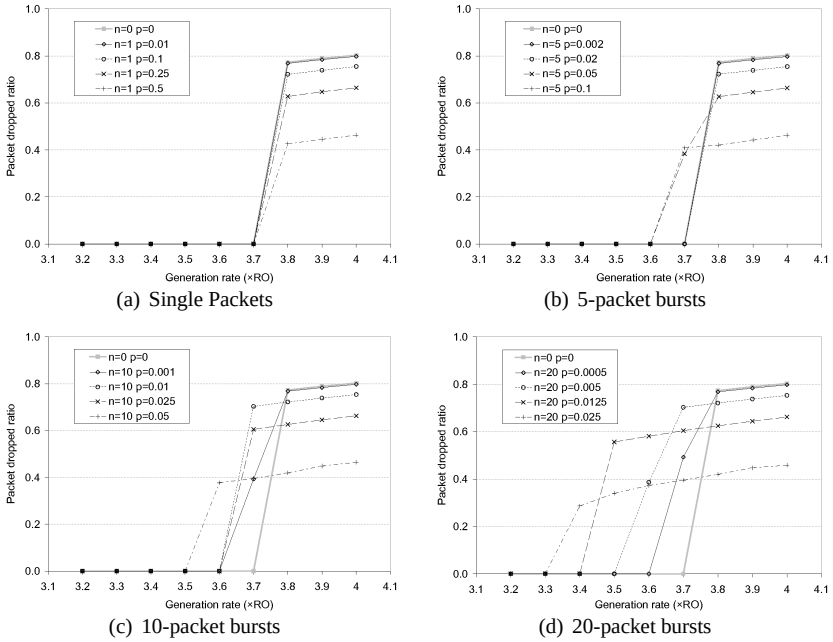


(a) Network fed at RO level (non-saturated) (b) Saturated network at 10×RO load

Fig. 8 Cumulative distance distribution functions measured at injection and consumption

These three distributions are almost identical when the system is handling expected loads (RO in Figure 8(a)) indicating that the network is not saturated. By contrast, when the system handles excessively high loads (10 RO in Fig. 8(b)), it reaches saturation and these distributions are very different. The distribution at injection shows no noticeable change compared with the previous scenario. However the distance distributions at consumption are noticeably different. D_c is shifted to the left (shorter distances), meaning that those packets that have to travel longer distances are more likely to be dropped. In contrast, D_t is shifted to the right (longer distances).

This is because an increase in the number of hops actually travelled by the packets reflects frequent activation of the emergency routing mechanism. Taken together, the locality and emergency routing tests suggest that good neural mappings should attempt to cluster routes towards local processor nodes. The performance figures suggest, however, that the neuron-to-node mapping, while being important, is not going to become a critical issue when simulating actual neural activity with SpiNNaker.



(a) Single Packets (b) 5-packet bursts

(c) 10-packet bursts (d) 20-packet bursts

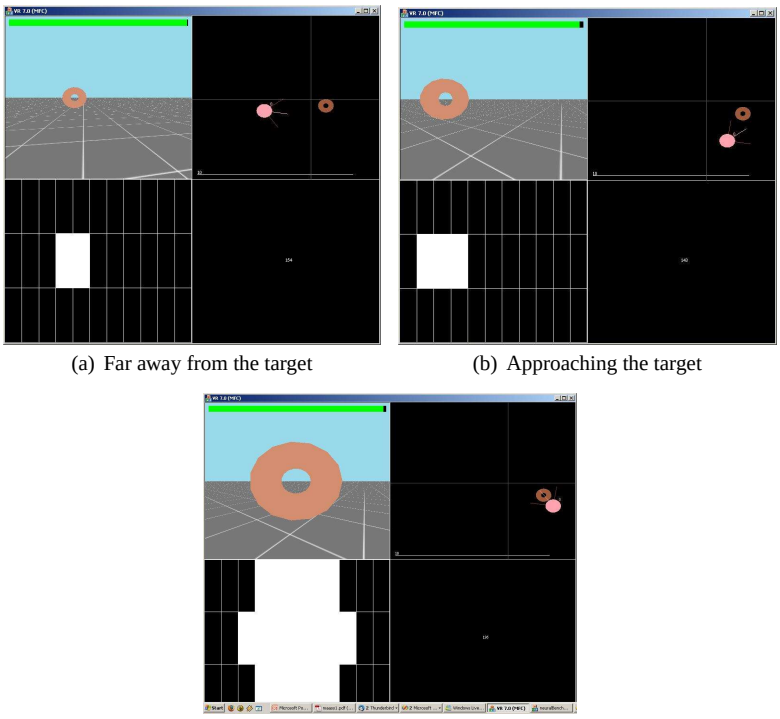
Fig. 9 Packet dropped ratio - 256x256 SpiNNaker network. Poisson traffic with causality. X and Y axes are as in fig. 7. Generation rates are shown in a shorter range for clarity.

Of more potential concern is the presence of burst traffic. The second set of experiments aims to measure the impact of traffic causality and burstines. Using the previously explained triggering mechanism we tested various configurations that manage the same *overall* amount of traffic but with different levels of causal generation of packets (i.e., G is fixed and i_r, p and n are selected accordingly). We fixed generation rates (G) around the point where independent-only traffic forced the system to drop packets (≈0.037 packets/cycle/node). Fig. 9 shows the results for the most distant traffic ($\lambda = 128$) but results are similar for other λ values when managing loads close to their corresponding saturation points. (Values $\lambda = 64$ and $\lambda = 32$ were checked, but not plotted for the sake of brevity.) In all cases, the higher degree of causality in the traffic, the lower the packet dropped ratio once the system reaches saturation. This is inherent to the causality of the traffic, because when packets are dropped they do not reach their destination; therefore they do not trigger other pack-

ets. For this reason the actual generation rate is lower than expected, which can be seen as a form of self-throttling of the workload. Notably, the larger the burst length and the probability to trigger a burst the lower the injection rate at which the network starts dropping packets, since large bursts generate congestion around the injecting node. Source-side management of packet generation: a function of the *software*, appears to be the more important factor in mitigating against packet loss than *hardware* limitations.

6.2 Simulation-Based Functionality Testing

Having established the traffic viability of the network, the next series of tests verify basic functionality: does the SpiNNaker chip faithfully reproduce the neural model? We performed tests both with the spiking model and the MLP model.



(a) Far away from the target (b) Approaching the target

(c) Target reached

Fig. 10 Doughnut hunter test. Successive frames show the network's "body" as it approaches the target.

6.2.1 Spiking Tests

We ran two different simulations using the spiking model. In the first we implemented a randomly-connected network with 48 excitatory and 16 inhibitory neurons having 40 connections per neuron with random 1-16 ms delay between neurons. We then stimulated 6 excitatory neurons and 1 inhibitory neuron of the population with a constant input in order to simulate external input. As we reported in [27] this network produced spiking patterns and synaptic learning consistent with that expected. In the second set of tests we created a synthetic environment: a "doughnut hunter" application. The network in this case had visual input and motion output; the goal was to get the position of the network's (virtual) body to a target: a toroid or "doughnut". Testing (figs. 10(a), 10(b), and 10(c)) verified that the network could successfully track and then move its body towards the doughnut, ultimately reaching the target. Although basic, these tests verified the functionality: the neural model behaved as expected both at the behavioural level and at the signal (spike) level.

6.2.2 MLP Tests

To test the MLP network we created an application based on the "digits" application from LENS (http://tedlab.mit.edu/~dr/lens), a software-based MLP simulator. Our network removed extraneous structural complications from the example to arrive at a simple feedforward network with 20 input, 20 hidden, and 4 output neurons. We trained the network using momentum learning with a momentum of 0.875 and a learning rate of 0.0078125, initialising the weights randomly between [-0.5, 0.5] We augmented the Lens-supplied data set with digits from 0-9 and added 2 sets of distorted digits with values 0-9. We then ran the network through 3 successive training epochs. Results are in fig. 11. Once again these results are consistent with basic functionality.

6.3 On-Chip Performance Testing

The final two tests used models running on the actual hardware - in this case on a PCB equipped with 4 SpiNNaker test chips interconnected as shown in fig. 12(a). Each chip connects to neighbouring chips over the corresponding physical link of the 6 available (the black lines in the figure). Test chips contain two cores, one for system-maintenance "monitor" functionality and the other one to simulate neurons: up to 1000 spiking units per core. The tests investigate the behaviour of a real multi-chip SpiNNaker system modelling networks of spiking neurons. We used LIF neurons [50] for these tests.

6.3.1 Synfire Chain

In order to test a scalable network and verify bursty network dynamics, we implemented a "synfire chain" model [1], [17]. A synfire chain is a feedforward neural network composed of groups of neurons or "pools", where every pool connects to the

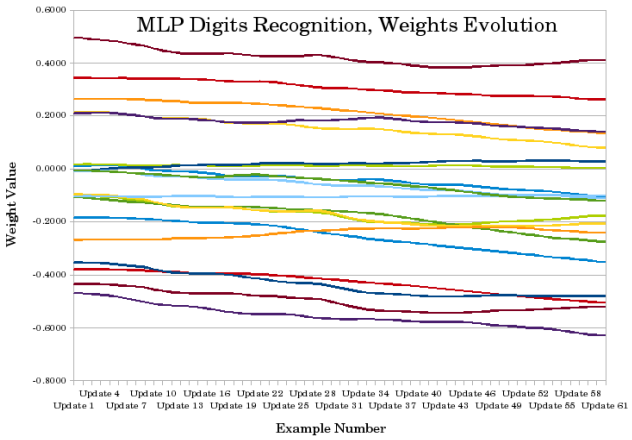


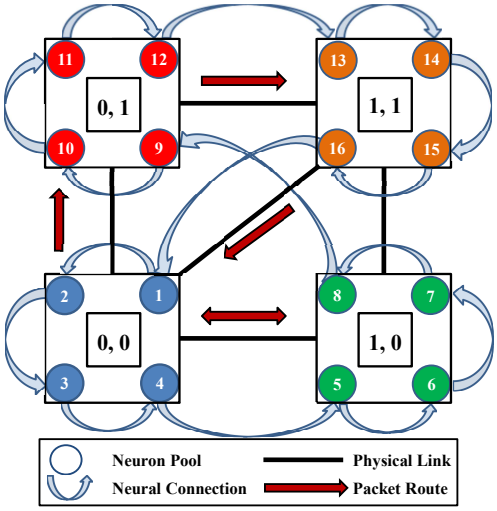
Fig. 11 SpiNNaker MLP test, weight changes. To improve readability the diagram shows only *selected* weights; unshown weights are similar. The weight changes show the expected evolution. Weight changes reflect an overall downward trend, consistent with early stages of momentum learning. The oscillations are characteristic of the learning rule.

next one in the hierarchy. Such a model propagates characteristic bursts of activity through the network. We simulated 4 pools composed of 250 neurons for each chip in a 4-chip testing environment, for a total of 16 pools and 4000 neurons (fig. 12(a)).

We number neuron pools from 1 to 16, each neuron in a pool connected to the corresponding neuron of the subsequent pool. The last pool is connected back to the first, providing inhibitory feedback. Weights are set so that one spike received will make the neuron fire, hence propagating the activity through the network. Presynaptic and post-synaptic neurons may reside on different chips (eg. connections from pool #4 and pool #5) or locally (eg. connections from pool #1 to pool #2). Red straight arrows indicate such routes. We stimulated 35 random neurons in the first population by injecting them with a current strong enough to make them fire at $\sim$20 Hz. The activity is then propagated with random delays in the range 1-8 msec through the other pools. Fig. 13 shows the results of the simulation, subdividing the raster plots and mean activity firing rate by chip. The firing rate is averaged over the number of neurons in the chip (1000), giving a mean population firing rate. Due to the nature of the network structure and simulation the activity is bursty, oscillating with peaks of 7 Hz (averaged over the whole population with a sliding window of 10 msec).

Table 12(b) shows spike activity and multicast (MC) packet counts for each chip. During the simulation every pool emits 756 spikes, for a total of 3024 spikes per chip. $\frac{3}{4}$ of the connections are local, implying that the same proportion of produced MC packets will be consumed locally (local to local packets), while $\frac{1}{4}$ will be routed off-chip. Every chip thus sends (local to external) and receives 756 (external to local) MC packets. Chip [0,0] also routes packets from pool #8 (chip [0,1]) to pool #9 (chip [1,0]), giving 756 external to external (transit) packets in the table for chip [0,0].

The on-chip network implementations use the PyNN [6], [10] multiplatform neural description environment, permitting direct comparison of the performance on-chip



(a) Configured Network Structure On Chip. Blue curved arrows depict neural connections between pools. Such connections are *virtual*, existing only at a model level; at the device level they correspond to routes over physical links (black lines).

Neurons per Pool:	**250**	**Pools per Chip: 4**
Neurons per Chip:	**1000**	**Total Neurons: 4000**
Spikes per Pool:	**756**	**Total Spikes: 12096**

	CHIP ID			
PACKET	**0,0**	**0,1**	**1,0**	**1,1**
local to local	2268	2268	2268	2268
external to external	756	0	0	0
external to local	756	756	756	756
local to external	756	756	756	756
TOTAL	**4536**	**3780**	**3780**	**3780**

(b) Spiking (multicast) packet activity summary

Fig. 12 Synfire network summary

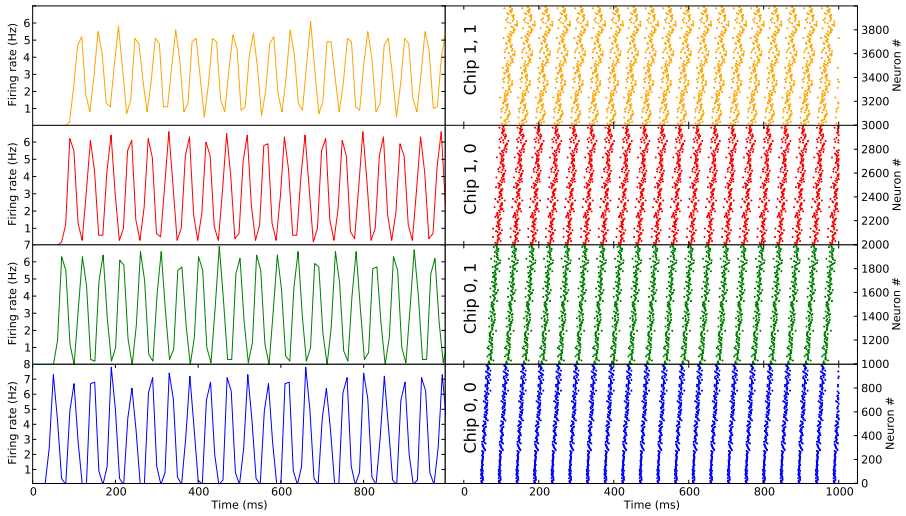


Fig. 13 Raster Plot

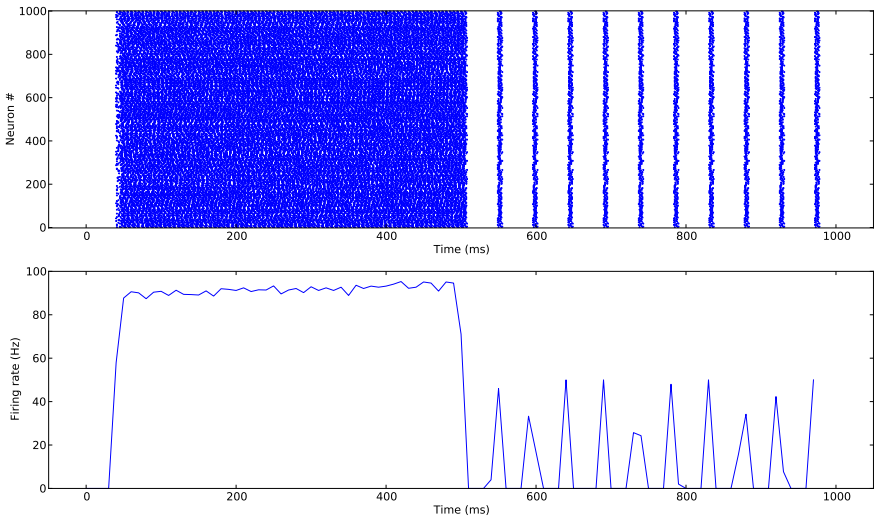


Fig. 14 Capacity limit testing: raster plot and mean firing rate for chip 0, 0. The firing rate is averaged over a time window of 10 ms.

with a standard software simulator - in this case NEST [11]. On-chip simulations run in real-time, thus the 1000 ms simulation time is the actual time to run. By comparison, a NEST implementation of the same model took 4163 ms to complete. While we hasten to emphasise that such a result is preliminary and based on a single observation, SpiNNaker thus demonstrates approximately a $4\times$ speedup at this (small) scale.

6.3.2 Capacity limits testing

In order to investigate the capacity of the inter-chip communication interface we populated each chip with 1000 neurons, connecting each neuron to the corresponding neuron in the next chip (one-to-one connection). The last chip feeds excitatorially back into the first. Every spike produced in a chip is thus sent to the next chip (i.e. there are no locally processed spikes). We stimulated 500 neurons in the first chip with a current sufficient to make them fire at $\sim$20 Hz. Positive feedback makes the activity build up up to $\sim$90 Hz producing $\sim$49,000 spikes in $\sim$500msec (fig. 14). The chip can sustain this level of activity for 500 msec but then the (software) communications buffer overflows, breaking the connection loop and leaving only the activity due to the input (vertical stripes in the raster plot after 500ms). This test demonstrates the ability of the communications infrastructure to sustain high activity rates for short periods of time, as well as the current limitations of the implemented mechanism. Further tests (not shown) indicated that the system could sustain continuous activity at lower frequencies, but exhibited rapid breakdown at higher frequencies.

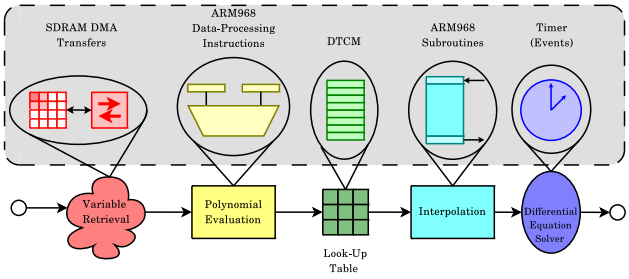


Fig. 15 A general event-driven function pipeline for neural networks. The grey box is the SpiNNaker realisation.

7 Discussion

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. We can define a generalised function pipeline that is adequate for most neural models in existence (fig 15). The pipeline 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. Meanwhile, SpiNNaker's processors can easily implement polynomial functions but other types, e.g. exponentials, are inefficient. In such cases it is usually easier to implement 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: thus, to solve such equations, it remains to pass them into a solver. For very simple cases it may be possible to solve them analytically, but for the general case, the Euler method evaluation we have used appears to be adequate.

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. Burstiness in the generation of traffic may generate contention around the node that is injecting. This contention may lead to dropping packets at loads at which the network would operate flawlessly with non-causal traffic, but still significantly higher than the load required during regular operation of the system. In mitigation, causal traffic tends to self-throttle, because dropping of packets leads to a reduction of the

packet generation rate. Another important discovery is that traffic burstiness affects the injection rate at which the network is forced to drop packets. The larger the burst length and the probability to trigger a burst are, the lower the generation rate at which the network starts dropping packets. This is because large bursts generate congestion around the injecting node, which can eventually spread to the whole network, forcing packets to be dropped at their *source*. We remark, however, that the simulated loads are more than three and a half times those required during regular operation of the system and that the spatial distribution of the traffic is utterly pessimistic. Careful analysis of the flow of execution on the SystemC model determined 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. 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 queue implementation. Clearly, very efficient interrupt service routines, together with aggressive source-side output management, are essential under extreme loading conditions.

Careful management of memory variables is also an important consideration. Both models involve multiple associative memories and lookup tables. If speed is critical, these must reside in DTCM or ITCM, and this places 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.

Solving differential equations introduces a third consideration: time efficiency and accuracy. Most nonlinear differential equations have no analytic solution, but numerical methods are computationally complex. The Euler method we used is usually an acceptable tradeoff, but it does introduce a synchronous element into the model. Furthermore the time step limits simulation accuracy. It also places fixed, absolute upper bounds on the computation time per neuron.

Both 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.

Notwithstanding these challenges, we have now demonstrated the ability of SpiNNaker, even in a reduced test chip configuration, to run models of reasonable size ($\sim$4000 neurons). This represents only a start, with simple models. We are currently working on implementing larger-scale, more biologically realistic models that simulate major subsystems of the brain and are scalable across a wide range of model sizes. Such models will include heterogeneous neuron and synaptic types operating with the same simulation, and possibly at different levels of structural abstraction. Part of this work includes the creation of more model types to expand system-level libraries, notably voltage-gated NMDA synapses with time-dependent channel kinet-

ics. Work on refining the packet processing, particularly in the host interface from SpiNNaker to the user, is also a major activity. We are conducting a systematic review and revision of the software model libraries, to streamline operation and improve "plug-in" development capability. This work provides improved support for full SpiNNaker chip, containing 18 cores and some enhanced features such as native support for atomic operations and debugging packets, which was delivered and began testing in May 2011. This chip will, obviously, support far larger models, and possibly more. There is evidence that in addition to neural models, SpiNNaker's parallel-processing architecture may find interesting uses outside the neural field, and thus we are investigating these where appropriate. Certainly, the emergence of such non-neural applications is an indication that SpiNNaker demonstrates important and possibly fundamental properties of parallel computing.

The pre-eminent feature 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. In other words, 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 describe its behaviour without having to consider how or even if a System Level service implements it. Structurally, 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.

8 Conclusions

By implementing an event-driven model directly in hardware, SpiNNaker comes considerably closer to biological neural computation than clocked digital devices. At the same time it brings into sharp relief the major differences from synchronous computation that place a much greater programming emphasis in event-driven computing 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. 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 paral-

lel 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. Most obvious is the use of (locally) synchronous ARM968 processors. Eventually it would be ideal to have each of the local programmable processors be themselves asynchronous. Meanwhile 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 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. These kinds of features would be asking for data corruption in a synchronous model but become logical in the event-driven model.

How far should neural network chips go in directly implementing the model in hardware? For years the mesmerising concept of "direct implementation" has been popular, yet it is fundamentally a misconception: since the "actual" model of computing in the brain is unknown, there can be no certainty a chip is directly implementing *anything*. The SpiNNaker neuromimetic architecture provides a more realistic and useful answer: instead of trying to answer the question, build systems that can define the problem.

Acknowledgements The SpiNNaker project is supported by the Engineering and Physical Sciences Research Council, partly through the Advanced Processor Technologies Portfolio Partnership at the University of Manchester, and through Grants EP/D07908X/1 and GR/S61270/01; and also by ARM and Silistix. When this research was performed Dr. Javier Navaridas was supported by a post-doctoral grant of the University of the Basque Country and is now a Newton International Fellow with the University of Manchester. Prof. Jose Miguel-Alonso is supported by the Spanish Ministry of Education and Science, grant TIN2010-14931, and by Basque Government grant IT-242-07. Dr. Mikel Luján holds a Royal Society University Research Fellowship. We appreciate the support of these sponsors and industrial partners.

References

1. Abeles, M.: Local cortical circuits : an electrophysiological study. Springer-Verlag (1982)
2. Ananthanarayanan, R., Modha, D.S.: Anatomy of a Cortical Simulator. In: Proc. 2007 ACM/IEEE Int'l Conf. on Supercomputing (SC'07), pp. 1–12 (2007)
3. Boahen, K.A.: 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 (2000)
4. Cauwenberghs, G.: An Analog VLSI Recurrent Neural Network Learning a Continuous-Time Trajectory. IEEE Trans. Neural Networks **7**(2), 346–361 (1996)
5. Dally, W.J., Seitz, C.L.: Deadlock-Free Message Routing in Multiprocessor Interconnection Networks. IEEE Trans. Computers **C-36**(5), 547–553 (1987)
6. Davison, A.P., Brüderle, D., Eppler, J., Kremkow, J., Muller, E., Pecevski, D., Perrinet, L., Yger, P.: PyNN: a common interface for neuronal network simulators. Frontiers in Neuroinformatics **2**(11) (2009)
7. Delorme, A., Thorpe, S.J.: SpikeNET: an event-driven simulation package for modelling large networks of spiking neurons. Network: Computation in Neural Systems **14**(4), 613–627 (2003)

8. Fieres, J., Schemmel, J., Meier, K.: Realizing biological spiking network models in a configurable wafer-scale hardware system. In: Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008), pp. 969–976. IEEE Press (2008)
9. Furman, B., White, J., Abidi, A.A.: CMOS Analog IC Implementing the Back Propagation Algorithm. Neural Networks **1**(Supplement 1), 381 (1988)
10. Galluppi, F., Rast, A., Davies, S., Furber, S.: A General-purpose Model Translation System for a Universal Neural Chip. In: Proc. 2010 Int'l Conf. Neural Information Processing (ICONIP 2010). Springer-Verlag (2010)
11. Gewaltig, M.O., Diesmann, M.: NEST (NEural Simulation Tool). Scholarpedia **2**(4), 1430 (2007)
12. Glackin, B., McGinnity, T.M., Maguire, L.P., Wu, Q.X., Belatreche, A.: A Novel Approach for the Implementation of Large Scale Spiking Neural Networks on FPGA Hardware. In: Proc. 8th Int'l Work Conf. Artificial Neural Networks (IWANN 2005), pp. 552–563. Springer-Verlag (2005)
13. Goodman, D., Brette, R.: Brian: a simulator for spiking neural networks in Python. Frontiers in Neuroinformatics **2**(5) (2008)
14. Graf, H.P., Hubbard, W., Jackel, L.D., de Vegvar, P.G.N.: A CMOS Associative Memory Chip. In: Proc. IEEE First Int'l Conf. on Neural Networks, pp. 461–468 (1987)
15. Hagmann, P., Cammoun, L., Gigandet, X., Meuli, R., Honey, C.J., Wedeen, V.J., Sporns, O.: Mapping the Structural Core of Human Cerebral Cortex. PLoS Biology **6**(7), 1479–1493 (2008)
16. Harkin, J., Morgan, F., Hall, S., Dudek, P., Dowrick, T., McDaid, L.: 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), pp. 483–486 (2008)
17. Hayon, G., Abeles, M., Lehmann, D.: A Model for Representing the Dynamics of a System of Synfire Chains. J. Computational Sci. **18**(1), 41–53 (2005)
18. Hines, M.L., Carnevale, N.T.: The NEURON simulation environment. Neural Computation **9**(6), 1179–1209 (1997)
19. Holler, M., Tam, S., Castro, H., Benson, R.: An Electrically Trainable Artificial Neural Network (ETANN) with 10240 "Floating Gate" Synapses. In: Proc. 1989 Int'l Joint Conf. Neural Networks (IJCNN1989), pp. 191–196 (1989)
20. Indiveri, G., Chicca, E., Douglas, R.: A VLSI Array of Low-Power Spiking Neurons and Bistable Synapses With Spike-Timing Dependent Plasticity. IEEE Trans. Neural Networks **17**(1), 211–221 (2006)
21. Izhikevich, E.: Simple Model of Spiking Neurons. IEEE Trans. on Neural Networks **14**, 1569–1572 (2003)
22. Izhikevich, E., Edelman, G.M.: Large-scale model of mammalian thalamocortical systems. Proc. National Academy of Sciences of the USA **105**(9), 3593–3598 (2008)
23. Izhikevich, E.M.: Which Model to Use for Cortical Spiking Neurons. IEEE Trans. Neural Networks **15**(5), 1063–1070 (2004)
24. James, M., Hoang, D.: Design of Low-Cost, Real-Time Simulation Systems for Large Neural Networks. J. Parallel and Distributed Computing **14**(3), 221–235 (1992)
25. Jin, X., , Luján, M., Khan, M.M., Plana, L.A., Rast, A.D., Welbourne, S.R., Furber, S.B.: Efficient Parallel Implementation of Multilayer Backpropagation Network on Torus-Connected CMPs. In: Proc. 2010 ACM Int'l Conf. on Computing Frontiers (CF'10), pp. 89–90 (2010)
26. Jin, X., Furber, S., Woods, J.: Efficient Modelling of Spiking Neural Networks on a Scalable Chip Multiprocessor. In: Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008) (2008)
27. Jin, X., Rast, A., Galluppi, F., Khan, M.M., Furber, S.: Implementing learning on the SpiNNaker universal neural chip multiprocessor. In: Proc. 2009 Int'l Conf. Neural Information Processing (ICONIP 2009). Springer-Verlag (2009)
28. Johansson, C., Lansner, A.: Towards cortex sized artificial neural systems. Neural Networks **20**(1), 48–61 (2007)
29. Khan, M., Lester, D., Plana, L., Rast, A., Jin, X., Painkras, E., Furber, S.: SpiNNaker: Mapping Neural Networks onto a Massively-Parallel Chip Multiprocessor. In: Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008) (2008)
30. Lazzaro, J., Wawrzynek, J., Mahowald, M., Silviotti, M., Gillespie, D.: Silicon Auditory Processors as Computer Peripherals. IEEE Trans. Neural Networks **4**(3), 523–528 (1993)
31. Lee, B.J., Sheu, B.W.: General-Purpose Neural Chips with Electrically Programmable Synapses and Gain-Adjustable Neurons. IEEE J. of Solid-State Circuits **27**(9), 1299–1302 (1992)
32. Lytton, W.H., Omurtag, A., Neymotin, S.A., Hines, M.L.: Just-in-Time Connectivity for Large Spiking Networks. Neural Computation **20**(11), 2745–2756 (2008)

33. Maguire, L., McGinnity, T.M., Glackin, B., Ghani, A., Belatreche, A., Harkin, J.: Challenges for large-scale implementations of spiking neural networks on FPGAs. Neurocomputing **71**(1–3), 13–29 (2007)
34. Mattia, M., Guidice, P.D.: Efficient Event-Driven Simulation of Large Networks of Spiking Neurons and Dynamical Synapses. Neural Computation **12**(10), 2305–2329 (2000)
35. Mehrtash, N., Jung, D., Hellmich, H., Schönauer, T., Lu, V.T., Klar, H.: Synaptic Plasticity in Spiking Neural Networks (SP^2INN): a System Approach. IEEE Trans. Neural Networks **14**(5), 980–992 (2003)
36. Migliore, M., Cannia, C., Lytton, W.W., Markram, H., Hines, M.L.: Parallel network simulations with NEURON. J. Computational Neuroscience **21**(2), 119–29 (2006)
37. Mouraud, A., Paugam-Moisy, H., Puzenat, D.: A distributed and multithreaded neural event driven simulation framework. In: Proc. IASTED Int'l Conf. Parallel and Distributed Computing and Networks, pp. 212–217 (2006)
38. Nageswaran, J.M., Dutt, N., Krichmar, J.L., Nicolau, A.: A configurable simulation environment for the efficient simulation of large-scale spiking neural networks on graphics processors. Neural Networks **22**(5–6) (2007)
39. Navaridas, J., Luján, M., Miguel-Alonso, J., Plana, L.A., Furber, S.B.: Understanding the Interconnection Network of SpiNNaker. In: Proc. 23rd Int'l Conf. Supercomputing (ICS'09), pp. 286–295 (2009)
40. Navaridas, J., Miguel-Alonso, J., Pascual, J.A., Ridruejo, F.J.: Simulating and evaluating interconnection networks with INSEE. Simulation Modelling Practice and Theory **19**(1), 494–515 (2011)
41. Navaridas, J., Plana, L.A., Miguel-Alonso, J., Luján, M., Furber, S.B.: SpiNNaker: Impact of Traffic Locality, Causality and Burstiness on the Performance of the Interconnection Network. In: Proc. 2010 ACM Conf. Computing Frontiers (CF'10), pp. 11–19 (2010)
42. Orellana, C.G., Caballero, R.G., Velasco, H.M.G., Aligue, F.J.L.: NeuSim: a modular neural networks simulator for Beowulf clusters. In: Proc. 6th Int'l Work-Conference on Artifical and Natural Neural Networks (IWANN 2001), Part II, pp. 72–79. Springer-Verlag (2001)
43. Pelayo, F.J., Ros, E., Arreguit, X., Prieto, A.: VLSI Implementation of a Neural Model Using Spikes. Analog Integrated Circuits and Signal Processing **13**(1–2), 111–121 (1997)
44. Plana, L., Furber, S., Temple, S., Khan, M., Shi, Y., Wu, J., Yang, S.: A GALS Infrastructure for a Massively Parallel Multiprocessor. IEEE Design & Test of Computers **24**(5), 454–463 (2007)
45. Porrmann, M., Witkowski, U., Kalte, H., Rückert, U.: Implementation of artificial neural networks on a reconfigurable hardware accelerator. In: Proc. 2002 Euromicro Conf. Parallel, Distributed, and Network-based processing, pp. 243–250 (2002)
46. Rast, A., Jin, X., Khan, M., Furber, S.: The Deferred Event Model for Hardware-Oriented Spiking Neural Networks. In: Proc. 2008 Int'l Conf. Neural Information Processing (ICONIP 2008). Springer-Verlag (2009)
47. Rast, A., Khan, M.M., Jin, X., Plana, L.A., Furber, S.: 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)
48. Rast, A., Welbourne, S., Jin, X., Furber, S.: Optimal Connectivity in Hardware-Targetted MLP Networks. In: Proc. 2009 Int'l Joint Conf. on Neural Networks (IJCNN2009), pp. 2619–2626 (2009)
49. Rast, A., Yang, S., Khan, M., Furber, S.: Virtual Synaptic Interconnect Using an Asynchronous Network-on-Chip. In: Proc. 2008 Int'l Joint Conf. on Neural Networks (IJCNN2008) (2008)
50. Rast, A.D., Galluppi, F., Jin, X., Furber, S.B.: The Leaky Integrate-and-Fire Neuron: A Platform for Synaptic Model Exploration on the SpiNNaker Chip. In: Proc. 2010 Int'l Joint Conf. Neural Networks (IJCNN2010), pp. 3959–3966 (2010)
51. Rast, A.D., Jin, X., Galluppi, F., Plana, L.A., Patterson, C., Furber, S.B.: Scalable Event-Driven Native Parallel Processing: The SpiNNaker Neuromimetic System. In: Proc. 2010 ACM Conf. Computing Frontiers (CF'10), pp. 20–29 (2010)
52. Rice, K.L., Vutsinas, C.N., Taha, T.M.: A Preliminary Investigation of a Neocortex Model Implementation on the Cray XD1. In: Proc. 2007 ACM/IEEE Int'l Conf. on Supercomputing (SC'07), pp. 1–8 (2007)
53. Ros, E., Carrillo, R., Ortigosa, E.M.: Event-Driven Simulation Scheme for Spiking Neural Networks Using Lookup Tables to Characterize Neuronal Activity. Neural Computation **18**(12), 2959–2993 (2006)
54. Rückert, U.: ULSI Architectures for Artificial Neural Networks. In: Proc. 9th Euromicro Wkshp. on Parallel and Distributed Processing, pp. 436–442 (2001)

55. Steinkraus, D., Buck, I., Simard, P.Y.: Using GPUs for machine learning algorithms. In: Proc. 8th Int'l Conf. Document Analysis and Recognition, pp. 1115–1120 (2005)
56. Upegui, A., Peña-Reyes, C.A., Sanchez, E.: An FPGA platform for on-line topology exploration of spiking neural networks. Microprocessors and Microsystems **29**(5), 211–223 (2005)
57. Vogelstein, R.J., Mallik, U., Vogelstein, J.T., Cauwenberghs, G.: Dynamically Reconfigurable Silicon Array of Spiking Neurons With Conductance-Based Synapses. IEEE Trans. Neural Networks **18**(1), 253–265 (2007)
58. Watts, L.: Event-driven simulation of networks of spiking neurons. In: Advances in Neural Information Processing (NIPS) 6, pp. 927–934. Morgan Kaufmann Publishers (1994)
59. Yasunaga, M., Masuda, N., Yagyu, M., Asai, M., Yamada, M., Masaki, A.: 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), pp. 527–535 (1990)
60. Yu, T., Cauwenberghs, G.: Analog VLSI Biophysical Neurons and Synapses With Programmable Membrane Channel Kinetics. IEEE Trans. Biomedical Circuits and Systems **4**(3), 139–148 (2010)