Scalable Event-Driven Native Parallel Processing: The SpiNNaker Neuromimetic System

Alexander D. Rast
School of Computer Science,
University of Manchester
Oxford Road
Manchester, UK
rasta@cs.man.ac.uk

Xin Jin
School of Computer Science,
University of Manchester
Oxford Road
Manchester, UK
jinxa@cs.man.ac.uk

Francesco Galluppi
Department of Psychology,
University of Rome
Via Dei Marsi, 78
Rome, Italy
francesco.galluppi@gmail.com

Luis A. Plana
School of Computer Science,
University of Manchester
Oxford Road
Manchester, UK
plana@cs.man.ac.uk

Cameron Patterson
School of Computer Science,
University of Manchester
Oxford Road
Manchester, UK
pattersc@cs.man.ac.uk

Steve Furber
School of Computer Science,
University of Manchester
Oxford Road
Manchester, UK
steve.furber@manchester.ac.uk

ABSTRACT

Neural networks present a fundamentally different model of computation from the conventional sequential digital model. Modelling large networks on conventional hardware thus tends to be inefficient if not impossible. Neither dedicated neural chips, with model limitations, nor FPGA implementations, with scalability limitations, offer a satisfactory solution even though they have improved simulation performance dramatically. SpiNNaker introduces a different approach, the "neuromimetic" architecture, that maintains the neural optimisation of dedicated chips while offering FPGA-like universal configurability. Central to this parallel multiprocessor is an asynchronous event-driven model that uses interrupt-generating dedicated hardware on the chip to support real-time neural simulation. While this architecture is particularly suitable for spiking models, it can also implement "classical" neural models like the MLP efficiently. Nonetheless, event handling, particularly servicing incoming packets, requires careful and innovative design in order to avoid local processor congestion and possible deadlock. Using two exemplar models, a spiking network using Izhikevich neurons, and an MLP network, we illustrate how to implement efficient service routines to handle input events. These routines form the beginnings of a library of "drop-in" neural components. Ultimately, the goal is the creation of a library-based development system that allows the modeller to describe a model in a high-level neural description environment of his choice and use an automated tool chain to create the appropriate SpiNNaker instantiation. The complete system: universal hardware, automated tool chain, embedded system management, represents the "ideal" neural modelling environment: a general-purpose platform that can generate an arbitrary neural network and run it with hardware speed and scale.

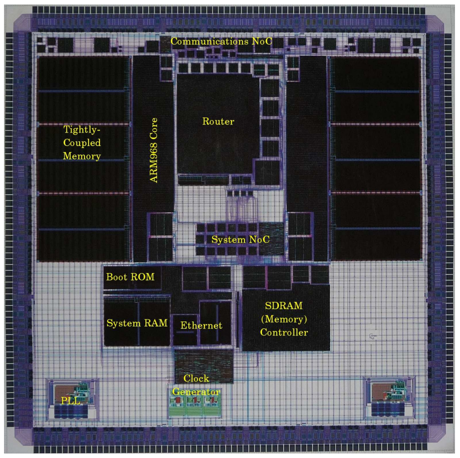

Figure 1: SpiNNaker test chip.

Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Neural nets*

General Terms

Design Performance Verification

Keywords

Asynchronous, event-driven, universal neural processor

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 [8] attempting biologically realistic simulation of brain-scale networks. While dedicated hardware is thus becoming increasingly attractive, it is also becoming clear that a fixed-model design would be a poor choice, 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. 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, an example to illustrate the differences between asynchronous parallel processing and parallel processing as it has been conceived in conventional sequential models.

2. NEURAL SYSTEM ARCHITECTURES

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 [7], through PC clusters [16] [18], to large mainframes [15]. 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 [5] and GENESIS (http://genesis-sim.org). Simulators like Brian [4] 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 or C/C++ 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 [26] [13] [1]. 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. 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. 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) [20] and graphics processor units (GPU's) [17]. 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. There have been 2 different reconfigurable approaches: component swapping [2] and network remapping [25]. Both seek to circumvent scaling limitations, with some success, but with both FPGA's and GPU's scalability limitations have proven to be the main problem, with FPGA's running into routing barriers due to their circuit-switched fabric [12] 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. This approach yields the greatest scope for architectural diversity as well as performance: different designs have used analogue or digital technology, hardwired or configurable architecture, continuous-activation or spiking signalling, coarse- or fine-grained 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 [6], chips use analogue circuitry to emulate, as closely as possible the actual biophysics of real neurons. The "neuroprocessor" approach [14], 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 topology. 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 circumvent such limitations while providing the scalable neural acceleration that only dedicated hardware can provide, we have introduced the SpiNNaker neuromimetic architecture.

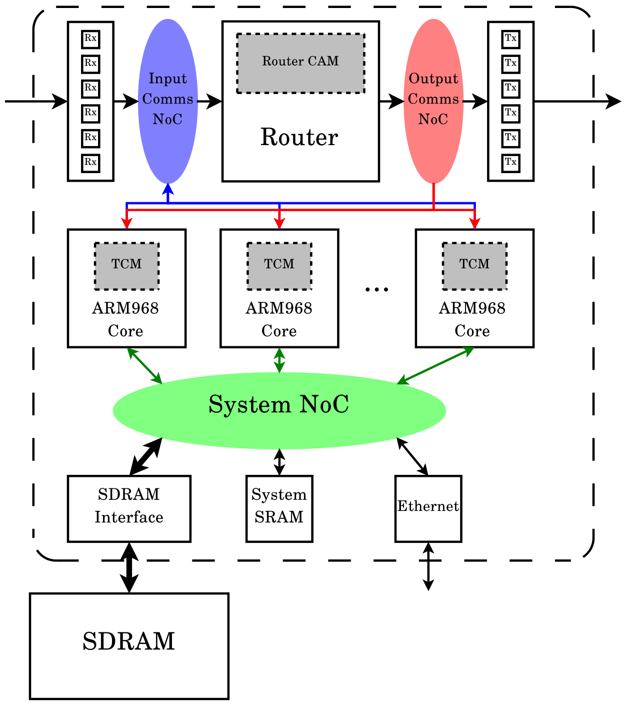

Figure 2: SpiNNaker Architecture. The dashed box indicates the extent of the SpiNNaker chip. Dotted grey boxes indicate local memory areas.

3. THE SPINNAKER NEUROMIMETIC IC

3.1 Implementation of the Neuromimetic Architecture

SpiNNaker implements the 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. Choice of the internal components reflects functionality that is useful specifically to neural networks.

3.1.1 Concurrent Multineuron Processing

SpiNNaker (fig. 2) contains multiple (2 in the present implementation, 20 in a forthcoming version) independent ARM968 processors, each simulating a variable number of neurons which could be as few as 1 or as many as 10,000. 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 Asynchronous Event-Driven Communications

SpiNNaker's communication network is a configurable packet-switched asynchronous interconnect using Address-Event Representation (AER) 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 ([19], [11]) 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 Global Memory

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 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, here grouped by postsynaptic neuron since incoming spikes carry presynaptic neuron information. This obviates the need for coherence checking because only one processor node will access a given address range. At the processor node level, the DMA controller handles synaptic data transfer, making the synapse appear virtually local to the processor by bringing it into DTCM when an incoming packet arrives [24]. 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 Reconfigurable Structure

SpiNNaker uses a distributed routing subsystem to direct packets through the Comms NoC. Each chip has a packet-switching router that can be reprogrammed in part or in full by changing the routing table, thus making it possible to reconfigure the model topology on the fly. Meanwhile, the DMA controller on the processing node can likewise swap out the running code on the processor in part or *in toto*, by copying data into the ITCM. With a small amount of irreplaceable code containing the interrupt handlers, it is then possible to alter dynamics, parameters, or virtually any other model characteristic in the middle of a simulation. Not only, therefore, is SpiNNaker capable of instantiating an arbitrary, user-defined user model, it can simulate neural models with dynamically changing structure, or even change model types entirely, in the middle of a simulation.

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 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, 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 [22] we introduced this 3-level software model for SpiNNaker, with a Model Level, a System Level, and a Device Level (fig. 3). 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. The important point to observe is: decisions about the event representation

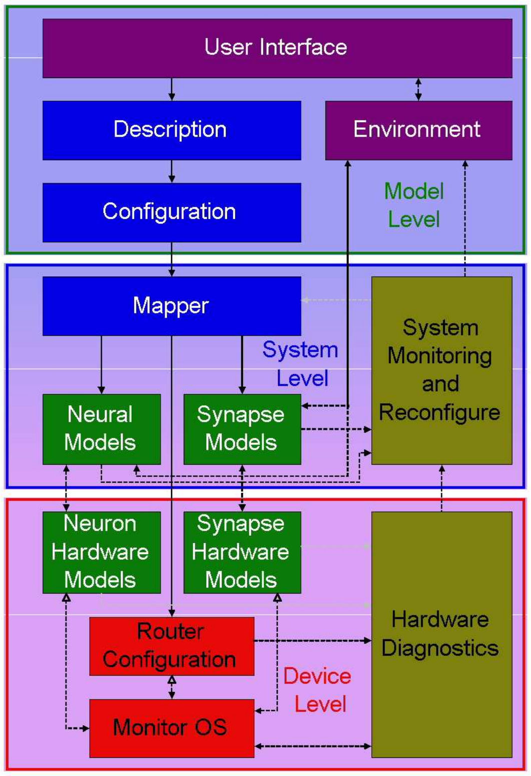

Figure 3: SpiNNaker Software Model.

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, it is better to have the Model level not define the event representation, but rather to have 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 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 methods or functions 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, the System level is mostly a collection of libraries for different neural models. Each library defines the event representation as set of source component 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. It would be possible to define the hardware sequences to set up event triggers and responses at the system level, but this would mean defining separate hardware sequences for each different model. This ignores the fact that much of the low-level hardware operation is common across all models and discards the possibility of reuse. Therefore the System level instead 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 will be 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. NEURAL MODEL IMPLEMENTATIONS

To test SpiNNaker functionality and performance, we have implemented 2 different neural network models, a spiking model and a classical MLP model. These models are sufficiently different in network design to represent an effective first test of SpiNNaker universality while sufficiently representative to be reference examples for future model implementations. We tested the models using ARM SoC Designer simulator, with additional low-level Verilog testing using Synopsys VCS.

5.1 Spiking neural network model

The first model is a spiking neural network using Izhikevich neurons and STDP synapses. The Izhikevich model [7] 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. Nonetheless, it should not be inferred that SpiNNaker was designed as a direct hardware implementation of the Izhikevich model. We describe many of the algorithmic details of this model in the following papers: [9] (Izhikevich model), [21], [10] (STDP implementation). 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.

5.1.1 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. As a result the process, operating at Device Level, uses the deferred-event model to schedule actual processing into the future as a System Level function. When an input arrives, the process performs an associative lookup on the packet address to find a source ID, corresponding to a row of synapses in memory. It then signals the DMA controller to perform a transfer of this row into the next-available of an array of synaptic buffers, incrementing the available buffer number. It then exits and returns control to the background.

5.1.2 DMA received

The DMA received and all other events are normal IRQ's, which means that in addition to their own processing they may need to account for the arrival of other packets. Notably in the case of DMA received, this means that its processing may not have completed before another DMA received event arrives, triggered by a packet arriving. The first task of the service routine at Device level therefore is to acknowledge the interrupt, thus 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 it triggers a System Level deferred service process operating in user mode rather than exception mode; otherwise it can simply return to the interrupted process (which will be a pre-existing deferred service). In its deferred service, operating at System Level, the process goes through the synaptic buffers in sequence. For each buffer, it first performs the STDP update, 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.

5.1.3 Timer

The Timer event has a higher priority than DMA received, since it operates on the current time rather than on future (delayed) time. Unlike the DMA-received 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) IRQ-mode operation consists of stacking registers and return addresses to the stack for a different mode: SVC (supervisor mode), acknowledging the interrupt, and changing to supervisor mode. By 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 STDP postsynaptic information. Efficiency is critical; the SVC-mode process must be able to complete long before the next (1ms) Timer interrupt which would have to kill any existing SVC process, corrupting neural and synaptic state.

5.2 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 [23], 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

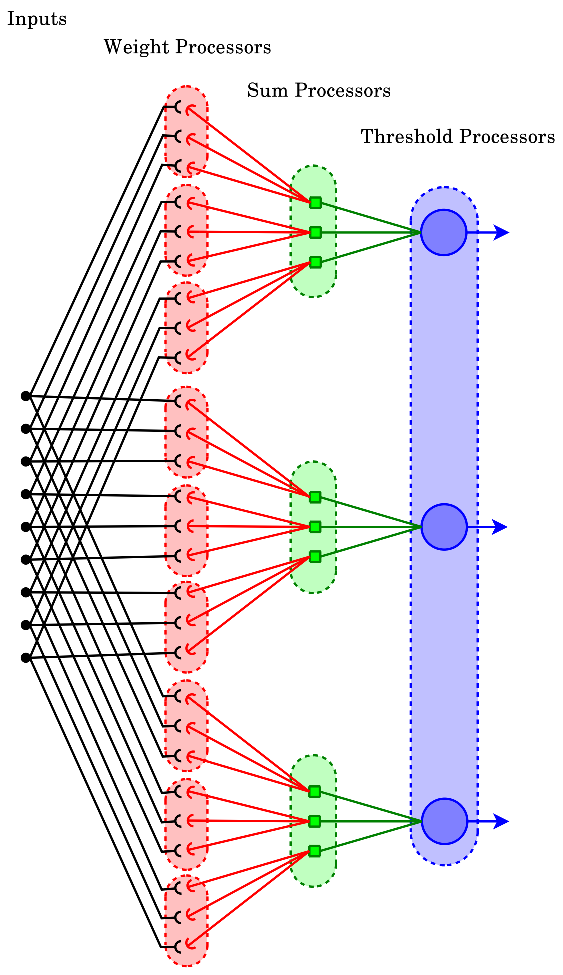

Figure 4: SpiNNaker MLP mapping. Each dotted oval is one processor. At each stage the unit sums the contributions from each previous stage. Note that one processor may implement the input path for more than one final output neuron (shown for the threshold stage here but not for other stages)

event representation for the dataflow. From this the process model will follow.

Because the mapping of the MLP to SpiNNaker distributes unit processing among several processors (see [23]) actual processing will vary depending on whether the processor in concern implements the weight, sum, or threshold part of a unit. 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. The inputs to weight units are the outputs from the previous layer of neurons. Each successive stage then represents a unit of aggregation, where the goal is to aggregate all the inputs to a given neuron j. Thus a weight unit will aggregate the inputs for one subgroup of all the inputs to neuron j. Sum processors are simply aggregation stages, accumulating the total contribution from multiple subgroups, so that a sum processor's inputs are one accumulated subgroup from the weight processor. Likewise, a threshold processor's inputs are one subgroup of sum processors (fig. 4). It is possible to cascade sum processors to create a neuron with arbitrary fan-in.

5.2.1 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 neuron ID (Technically, a "unit" instead of a neuron: in the MLP the unit is a processing element not necessarily associated with a single neuron) and uses the payload to transmit that unit's activation. Defining the events in the MLP model comes from the dataflow. 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, corresponding to the packet received event. 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 on output sent. This would be an Internal event. 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.

5.2.2 Packet Received

The component 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 time.

5.2.3 *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 process that toggles the mode between forward and backward. Changing direction itself is a simple task, but detecting the condition "all inputs arrived" requires consideration. 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. 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.

6. SIMULATION RESULTS

We ran simulations with 3 major objectives: chip verification, confirmation of accurate heterogeneous model support, and packet processing performance evaluation. Simulations used ARM SoC Designer Simulator on a complete SystemC model of the SpiNNaker chip. Our simulation implemented a 4-chip system with 2 processors per chip (corresponding to the first test board).

6.1 Functionality testing

The first and most important 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.

6.1.1 *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 [10] 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: an annulus or "doughnut". Testing (figs. 5, 6, and 7) 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.

6.1.2 *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

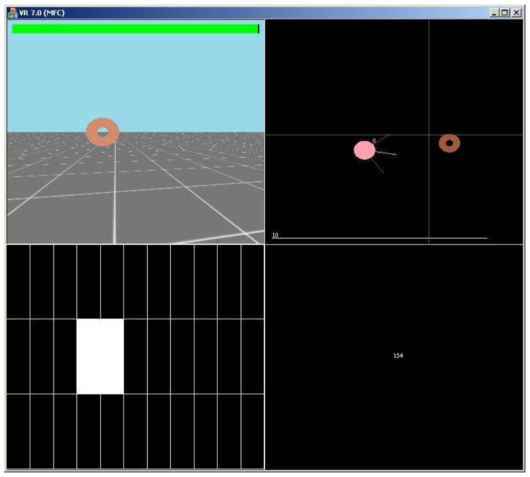

Figure 5: Far away from the target

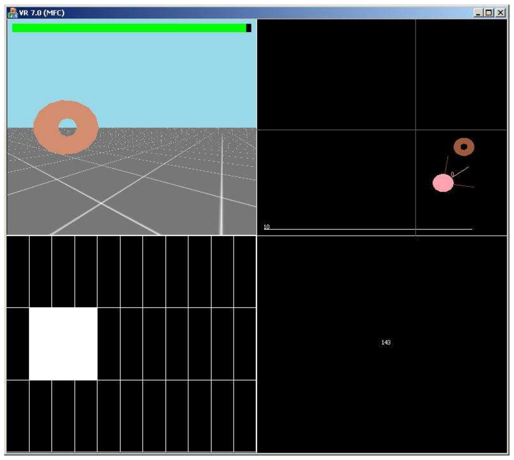

Figure 6: Approaching the target

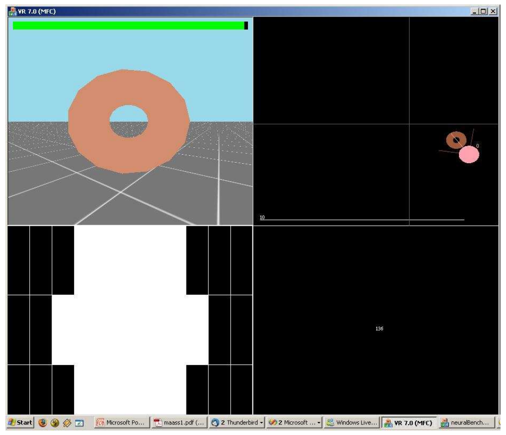

Figure 7: Doughnut hunter test. Successive frames show the network's "body" as it approaches the target. Above is when the network reaches the target.

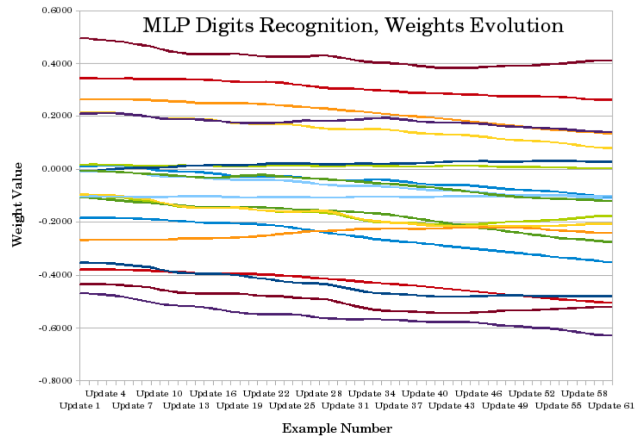

Figure 8: SpiNNaker MLP test, weight changes. To improve readability the diagram shows only *selected* weights; unshown weights are similar. The weight changes show the expected evolution. Weight changes reflect an overall downward trend, consistent with early stages of momentum learning. The oscillations are characteristic of the learning rule.

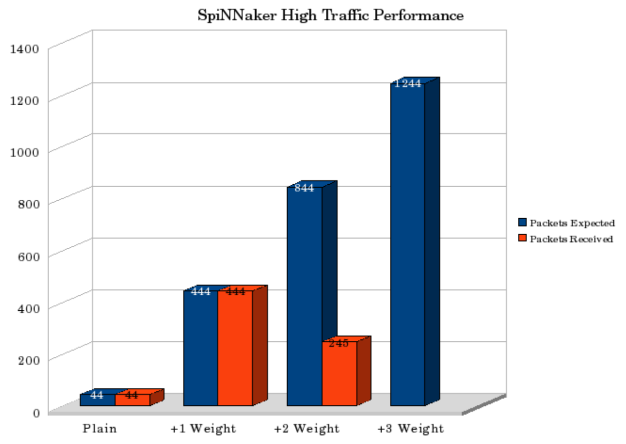

Figure 9: SpiNNaker packet handling performance

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. 8. Once again these results are consistent with basic functionality.

6.2 Packet Performance Testing

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. We used the MLP model as a test case for packet congestion by generating additional packets from the actual traffic of the simulation proper. To do this, we generated a separate packet type that output the weight of each synapse, for each input. We 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. 9. 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.

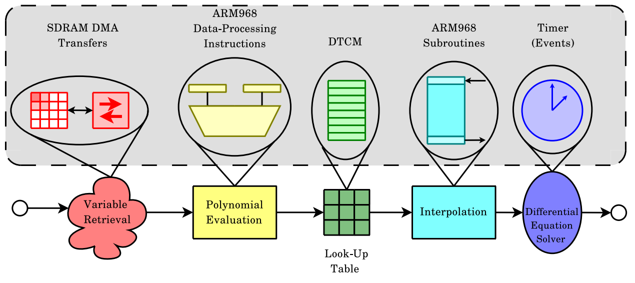

Figure 10: A general event-driven function pipeline for neural networks. The grey box is the SpiNNaker realisation.

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 10). 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. 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. 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.

There remains considerable work to be done. Since SpiNNaker hardware is now available, testing the models on the physical hardware is an obvious priority, along with testing larger and more complex 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. Part of this work also includes the creation of more model types to expand system-level libraries, notably a leaky-integrate-and-fire neuron and voltage-gated NMDA synapses. Work on refining the packet processing, particularly in the host interface from SpiNNaker to the user, is also a major activity. There is evidence that in addition to neural models, SpiNNaker's parallel-processing model 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 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. 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 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. 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.

9. ACKNOWLEDGMENTS

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 also by ARM and Silistix. Steve Furber holds a Royal Society-Wolfson Research Merit Award. We appreciate the support of these sponsors and industrial partners.

10. REFERENCES

[1] A. Delorme and S. J. Thorpe. “SpikeNET: an event-driven simulation package for modelling large networks of spiking neurons”. *Network: Computation in Neural Systems*, 14(4):613–627, November 2003.

[2] B. Glackin, T. M. McGinnity, L. P. 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. Artificial Neural Networks (IWANN 2005)*, pages 552–563. Springer-Verlag, 2005.

[3] D. Goldberg, G. Cauwenberghs, and A. Andreou. “Analog VLSI spiking neural network with address domain probabilistic synapses”. In *Proc. 2001 IEEE Int’l Symp. Circuits and Systems (ISCAS2001)*, pages 241–244. IEEE Press, 2001.

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

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

[6] G. Indiveri, E. Chicca, and R. Douglas. “A VLSI Array of Low-Power Spiking Neurons and Bistable Synapses With Spike-Timing Dependent Plasticity”. *IEEE Transactions on Neural Networks*, 17(1):211–221, Jan. 2006.

[7] E. Izhikevich. “Simple Model of Spiking Neurons”. *IEEE Trans. on Neural Networks*, 14:1569–1572, Nov. 2003.

[8] E. 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.

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

[10] 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.

[11] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras, and S. Furber. “SpiNNaker: Mapping Neural Networks onto a Massively-Parallel Chip Multiprocessor”. In *Proc. 2008 Int’l Joint Conf. on Neural Networks (IJCNN2008)*, 2008.

[12] L. 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(1–3):13–29, December 2007.

[13] M. Mattia and P. D. Guidice. “Efficient Event-Driven Simulation of Large Networks of Spiking Neurons and Dynamical Synapses”. *Neural Computation*, 12(10):2305–2329, October 2000.

[14] N. Mehrtash, D. Jung, H. Hellmich, T. Schönauer, V. T. Lu, and H. Klar. “Synaptic Plasticity in Spiking Neural Networks (SP^{2}INN): a System Approach”. *IEEE Transactions on Neural Networks*, 14(5):980–992, Sept. 2003.

[15] 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.

[16] A. Mouraud, H. Paugam-Moisy, and D. Puzenat. “A distributed and multithreaded neural event driven simulation framework”. In *Proc. IASTED Int’l Conf. Parallel and Distributed Computing and Networks*, pages 212–217, 2006.

[17] 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.

[18] C. G. Orellana, R. G. Caballero, H. M. G. Velasco, and F. J. L. Aligue. “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*, pages 72–79. Springer-Verlag, 2001.

[19] L. Plana, S. Furber, S. Temple, 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, Sept.-Oct. 2007.

[20] 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.

[21] 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)*. Springer-Verlag, 2009.

[22] A. Rast, M. M. Khan, X. Jin, L. A. Plana, and S. 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.

[23] A. Rast, S. Welbourne, X. Jin, and S. Furber. “Optimal Connectivity in Hardware-Targetted MLP Networks”. In *Proc. 2009 Int’l Joint Conf. on Neural Networks (IJCNN2009)*, pages 2619–2626, 2009.

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

[25] 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.

[26] L. Watts. “Event-driven simulation of networks of spiking neurons”. In *Advances in Neural Information Processing (NIPS) 6*, pages 927–934. Morgan Kaufmann Publishers, 1994.