Algorithm and Software for Simulation of Spiking Neural Networks on the Multi-Chip SpiNNaker System

Xin Jin, Francesco Galluppi, Cameron Patterson, Alexander Rast, Sergio Davies, Steve Temple, and Steve Furber

Abstract— **This paper presents the algorithm and software developed for parallel simulation of spiking neural networks on multiple SpiNNaker universal neuromorphic chips. It not only describes approaches to simulating neural network models, such as dynamics, neural representations, and synaptic delays, but also presents the software design of loading a neural application and initial a simulation on the multi-chip SpiNNaker system. A series of sub-issues are also investigated, such as neuron-processor allocation, synapses distribution, and route planning. The platform is verified by running spiking neural applications on both the SoC Designer model and the physical SpiNNaker Test Chip. This work sums the problems we have solved and highlights those requiring further investigations, and therefore it forms the foundation of the software design on SpiNNaker, leading the future development towards a universal platform for real-time simulations of extreme large-scale neural systems.**

I. Introduction

Parallel simulations of biological-realistic spiking neural network models have been of much interest to computational neuroscientists to tackle the constrain of simulation speed. Many different attempts have been made towards this objective, using supercomputer-based systems, such as IBM Blue Gene/L [1] or Beowulf cluster [2]. These systems are powerful and easy to use, in the cost of large power consumption and the huge size, which in turn make these solutions difficult to be applied in the filed of embedded intelligent systems such as robots. Dedicated hardware solutions, based on FPGAs [3], analogue circuits [4] or hybrid analog-digital VLSI [5], lacks flexibility in choosing neuronal dynamics, synapse models or learning rules. Due to the experimental nature of the neural modeling caused by the "unknowns" in neuroscience, neuromorphic hardware needs not be hardwired to a specific neural model to provide flexibility, but should be also capable of delivering high computational power.

The SpiNNaker system is proposed in this context. It is a scalable massive parallel computing system under development with the aim of building a general-purpose platform for the real-time parallel simulation of large-scale spiking neural systems [6], [7]. Each SpiNNaker chip contains 20 identical ARM968 subsystems to provide the processing power and are flexible enough to modeling a variety of neuronal dynamics and synapses. The 6 external links associated with each chip allow chips to be arranged in a hexagonal mesh with bidirectional connections to 6 neighbors. The multi-cast mechanism, self-timed asynchronous inter-connections, and the distributed on-chip and off-chip memory system, together allow virtual synaptic connections which enable the ability of re-wire neural connections [8]. The parallel simulation of neural networks on SpiNNaker provides the possibility to guarantee the necessary processing power by adding more chips into the system when the scale of the neural network simulated increases.

For such a system, well-defined modeling algorithm needs be to investigated for efficient mapping neural networks. An easy-to-use software system is also required for target users to instantly configure the system and run their existing neural models. This paper explores solutions to these issues by going through the flow to run spiking neural applications on both a four-chip SpiNNaker SoC Designer model [9] and a single physical SpiNNaker Test Chip. Depending on their own features, the feasibility of applying spike-based learning algorithms have to be investigated individually. A good example of implementing spike-timing dependant plasticity (STDP) on SpiNNaker can be found in [10].

The rest of the paper is organized as following: The approaches to modeling the Izhikevich neuronal dynamics, neural representations, and delays are presented in Section II and III. The event-driven scheduler used to organize the different tasks and events is described in Section IV. Solutions for and results from the multi-chip simulation of spiking neural network applications are presented in Section V. Related issues are discussed in Section VII which is followed by a conclusion in the final section.

II. Modeling neuronal dynamics

SpiNNaker has been developed to be a generic neural simulation platform, which in turn requires to support multiple neuron models. The neural dynamical models describe the neuron behavior in response to input stimuli and the process of producing output spikes. The model itself is usually stand alone and independent from other parts of the neural network, for instance the connectivity, coding scheme and learning. Since neurons are only simulated on processors, it decouples the implementation of neuronal dynamical model from the implementation of the network. This allows us to switch easily between different neuronal models – either conductance- or phenomenological-based.

For this prototype system, the Izhikevich model [11] is chosen as an example for real-time simulation with 1 ms resolution. 16-bit fixed-point arithmetic is used in the implementation to save both processing power and storage space.

The authors are with the School of Computer Science, The University of Manchester, Manchester, UK (email: {jinxa}@cs.man.ac.uk)

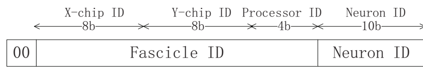


Fig. 1. The routing key

A dual-scaling factor scheme, which applies two different scaling factors to convert floating- to fixed-point numbers (large floating numbers with a small scaling factor and small floating numbers with a large scaling factor), is used to reduce the precision lost during the conversion [12]. The performance is also optimized by:

1) Expanding the width from 16-bit to 32-bit during the computation to achieve better precision.
2) Transformation of the equations to allow using some efficient ARM specific instructions.
3) Adjusting the parameters and doing as much pre-computation as possible.
4) Programming in assembler language.

In our implementation, one iteration of Izhikevich equations takes 6 fixed-point mathematical operations plus 2 shift operations, which is more efficient than the original implementation which takes 13 floating-point operations [11]. In a practical implementation, the whole subroutine for Izhikevich equations computation can be performed by as few as about 20 instructions if the neuron does not fire, otherwise, it takes about 10 more instructions to reset the value and send a spike event. The detailed implementation, precision and performance analysis can be found in [12].

III. Modeling neural representations using EAM scheme

The high density of one-to-many (high fan-out) transmissions of packets in neural network, leads to inefficient communication on conventional parallel hardware. To achieve low communication overhead, an event-address mapping (EAM) scheme is used on SpiNNaker [12].

A. Spike propagation

The EAM scheme keeps synaptic weights at the post-synaptic end and set a relationship (in the lookup table) between the spike event and the address of the synaptic weight, hence no synaptic weight information needs to be carried in a spike event. When a neuron fires, *identical* packets, also referred to as routing keys with the format shown in Figure 1, are sent by efficient multicast to post-synaptic neurons. Each packet propagates through a series of multicast routers according to the pre-loaded routing table in each router, and finally arrives at the destination fascicle processors.

Each router entry contains a key value, a mask and an output vector. When a packet arrives, the routing key encoded in the packet is compared with the key in each entry of the MC table, after being ANDed with the mask. If it matches, the packet is sent to the ports contained in the output vector of this entry, otherwise, the packet is sent across the router by default routing, normally via the port opposite the input.

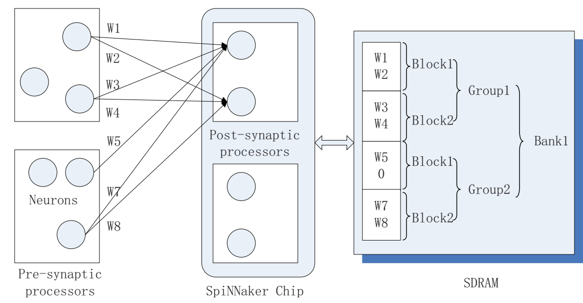


Fig. 2. The hierarchical structure of synaptic weight storage, where each circle denotes a neuron.

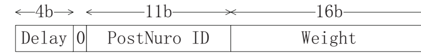


Fig. 3. The Synaptic Word.

The routing operation is performed using parallel associative memory for matching efficiency.

B. Synaptic weight storage

The EAM scheme employs two memory systems, Data-Tightly Coupled Memory (DTCM) and SDRAM, for efficient storing synaptic connections. DMA operations are used to transfer each Weight Block from the SDRAM to the local DTCM, before processing.

The SDRAM used for synaptic weight storage has the hierarchical structure as shown in Figure 2:

- The memory comprises a number of Banks, each is associated with one of the processors (while rectangle) on the chip.
- Each Bank comprises a number of Groups, each contains the synaptic weights of all connections from a source fascicle to this local fascicle.
- Each Group comprises a number of Blocks, each contains a number of Synaptic Words for all connections from one pre-synaptic neuron (on the corresponding source fascicle) to post-synaptic neurons in this fascicle, a one-to-many connection scenario.
- Each Synaptic Word represents one synapse, as shown in Figure 3 (“0” is a bit not used).

C. Finding the synaptic Block

A processor can easily find the associated synaptic weights to the fired neuron by matching the incoming spike packet with entries in the routing table, as shown in Figure 4. The lookup table organized in a binary tree maintains the mapping between the routing key and the Synaptic Block in the SDRAM.

D. Modeling the delay and input current

Biologically, a spike generated by a pre-synaptic neuron takes time measured in milliseconds to arrive at a post-synaptic neuron. In an electronic system, the communication delay is uncertain depending on the distance and the network workloads. Electronic delays, however, ranging

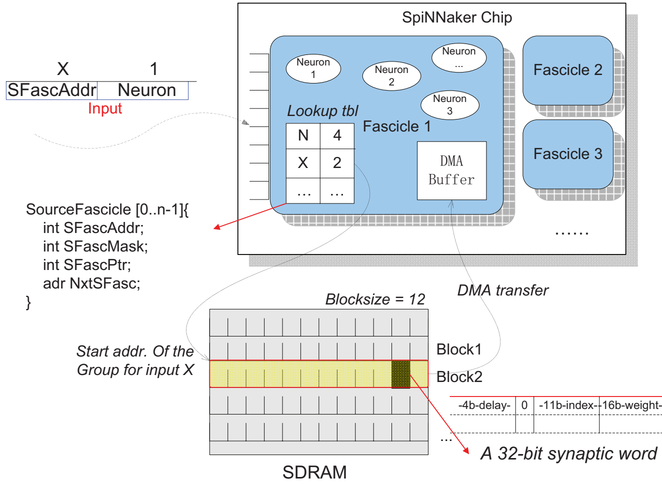


Fig. 4. Finding the synaptic Block.

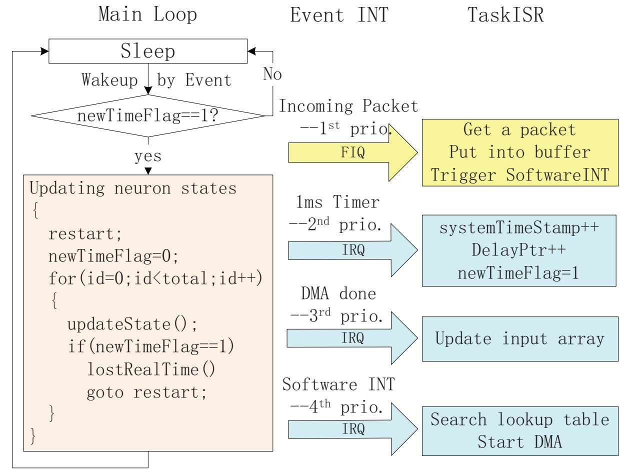


Fig. 5. The event driven model.

from nanoseconds to microseconds, are much smaller than biological delays. Hence the electronic delay can be ignored, and the packet arrival time can be considered as the neuron firing time. However, delays play an important role in neural modeling. Hence it is necessary to put delays back into the system, but using another approach, to enforce an agreement between biological and electronic time. The four most significant bits are used in a Synaptic Word to represent the delay of the connection, allowing us to simulate a delay up to 15 ms with 1 ms resolution. The electronic packet arrival time must be increased by a delay value to generate the real "biological" time, before applying the "weight update" process.

Weights in the Synaptic Block transferred to the DTCM will be loaded into an 16 elements circular input array (corresponding to synaptic delays of 0 ms – 15 ms). Each element represents the amplitude of all inputs applied to this neuron at a certain time. A pointer "DelayPtr" points to the element at the current time (0 ms) and moves forward one element per 1 ms with wraparound after 15 ms. The 4-bit delay is fetched and used to determine which element the weight will be updated into. The updating of the weight is reconfigurable and can be designed according to the type of synapse. In this system, a linear accumulate operation is used, in accordance with the synapse model used in most current neural network models.

IV. Scheduler

An event-driven model is proposed to schedule the multi-task system, as shown in Figure 5:

1) Incoming packet event is set to the first priority to receive a packet as soon as possible to avoid congestion, and triggers a software interrupt.
2) 1ms Timer event is set to the second priority and assigns values to several global variables: systemTimeStamp (the number of millisecond simulated) is increased by 1; the delay pointer DelayPtr is increased by 1 to point at the next element in the input array; newTimeFlag is set to indicate that the system is moving into a new millisecond.
3) The DMA done event is set to the third priority and performs updating of input array after the Synaptic Block has been transferred from the SDRAM to the DTCM.
4) The software INT event is set to the fourth priority. It retrieves a packet from the communication buffer and then processed to start the DMA operation.
5) The "updating neuron states" task is performed in the main loop. If all neuron state have been updated, the processor goes to sleep. The sleep mode is interrupted by an event to execute the corresponding ISR. Then the processor either performs the "updating neuron states" task or goes back to sleep mode, depending on the newTimeFlag status. The "updating neuron states" task firstly resets the newTimeFlag, and then updates neuron states. If newTimeFlag is detected as set during the updating, the task will be interrupted compulsorily to skip the remaining neurons, and a new iteration will be started.

V. Multi-processor simulation

SpiNNaker chip can be easily expanded to support neural networks in larger-scale. In this section we investigate the approach to run a multi-processor/chip simulation. A series of issues involved are investigated. Algorithms and software (called "InitLoad") are developed to solve them.

A. Input and output files

As shown in Figure 6, InitLoad provides two options: a random mode and a pre-defined mode. The random mode generates a random neural network according to statistic parameters; the pre-defined mode loads a user defined neural network. They both have three files to describe the system: "*SpiNNaker.ini*", "*Neurons. txt*", and "*Connections.txt*". The SpiNNaker.ini file is used to describe a SpiNNaker system, and the basic information of the neural network. Neurons.txt

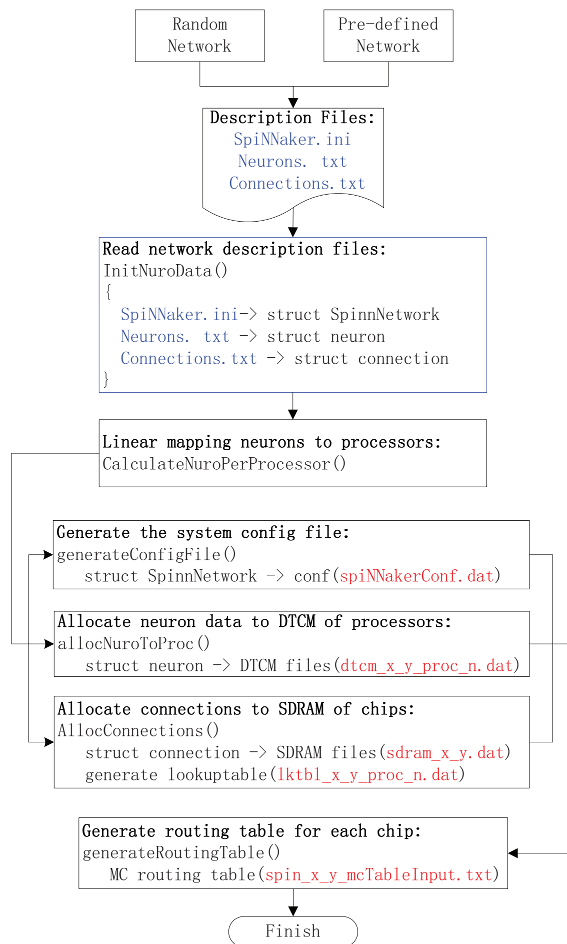


Fig. 6. The InitLoad flow chart

file contains a list of the neurons composing the network with their specific parameters. Connections.txt contains the neuron connection information. The description file can also be easily extended using some standard format such as XML.

The input files are loaded and output files, including neuron data files, synaptic weights files, lookup table files and routing table files, are produced according the mapping rule previously discussed.

1) Chip IDs: SpiNNaker chips have wrap-around connections (or they can be disabled to create a non-toroidal model); thus it is always possible to take the host chip (attached to the Host PC) as the origin (0,0), and the rest of the chips sitting in the first quadrant (I) can be indexed accordingly using (x, y) as shown in Figure 7.

2) Linear neuron-processor mapping: A linear mapping algorithm is used, where neurons are uniformly allocated to processors in order:

- SpiNN.nuroPerFasc = SpiNN.numNuro/(numfasc-1)
- SpiNN.nuroPerFasc = SpiNN.numNuro%(numfasc-1)

B. Synapse allocation

Synapses (connections) kept in the SDRAM need to be distributed to every chip according to the linear mapping

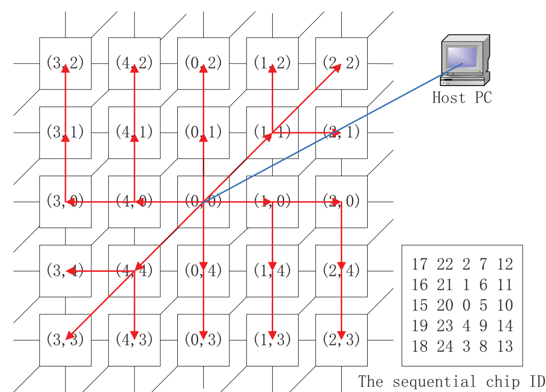


Fig. 7. Route planning

and the EAM scheme. The AllocConnections() function in InitLoad is designed to handle this task. Meanwhile, the lookup tables associated will also be created by this function.

C. Routing table generation

Routing tables are generated by the generateRoutingTable() function, according to the neuron connections. The routing is planned based on the order of the pre-synaptic neurons.

1) Routing table compression: Initially, a raw routing table will be generated, which simply stores one route for one pre-synaptic neuron and thus it is very big. However, what usually happens is that most packets from the neurons in the same pre-synaptic fascicle will be sent through the same routes. In this case, the MASK of the entry can be set properly to compress the routing table and reduce the number of entries. A third-party Logic Minimization Software – Espresso is used for the compression.

2) Route planning: The route planning algorithm we used guarantees to find a short path between two neurons as shown in Figure 7. Assuming a packet is transmitted by a neuron in chip (0,0), if the destination is in the first quadrant (I) or the third quadrant (III), the packet will firstly be sent in the diagonal direction. When it reaches the chip with the same x-coordinate or y-coordinate as the destination chip, it then goes straight towards the destination following the x-coordinate/y-coordinate vertically/horizontally. If the destination is in the second quadrant (II) or fourth quadrant (IV), the packet firstly goes horizontally following the x-coordinate. When it reaches the chip with the same x-coordinate as the destination chip, the packet then goes vertically towards the destination following the y-coordinate. The findMcRouting() function is used for route planning.

VI. Simulation

A. Initialization

We verify the implementation based on a four-chip SpiNNaker SoC Designer model [9] as shown in Figure 8. The application code and neural data need to be downloaded to SpiNNaker chips to start the simulation. The semihosting functions provide by SoC designer allow users to access files

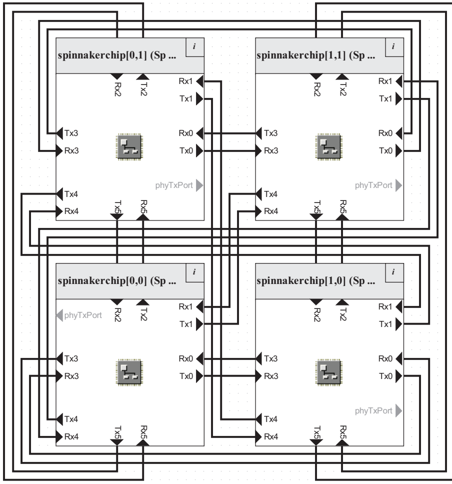


Fig. 8. A four chip model in SoC Designer

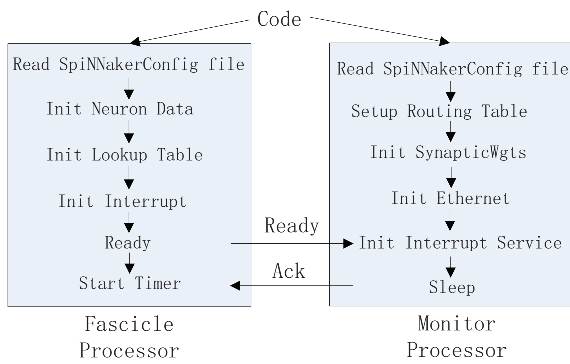
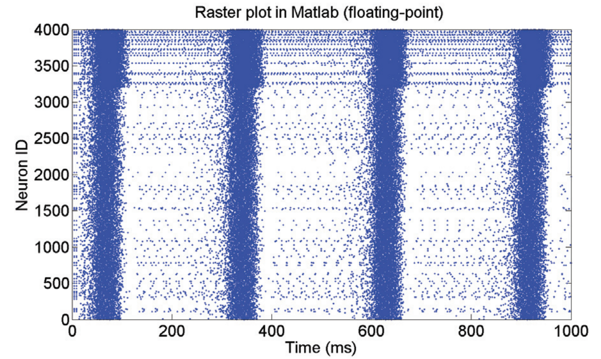


Fig. 9. The fascicle and monitor processors during initialization.

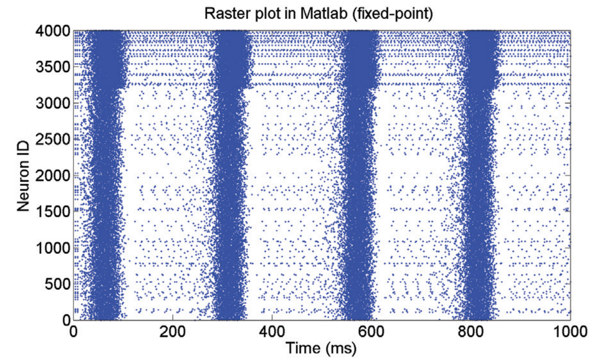
on the hard disk of the Host PC directly, making the code and data downloading easy. Dumping simulation results (spike raster, single neuron activity, and so on) is also easy during the simulation. The output data is written to files on the Host PC and plotted in the Matlab.

There are two processors (the same number of processors as in the SpiNNaker test chip), a Fascicle and a Monitor Processor, in each chip of the four-chip model. The Fascicle and the Monitor Processor load the same code at startup, as shown in Figure 9, each processor then performs a series of initialization processes. The Fascicle Processor mainly loads its private data set, while the Monitor Processor loads public data, such as synaptic weights and routing tables, which is shared by all the Fascicle Processors on the same chip.

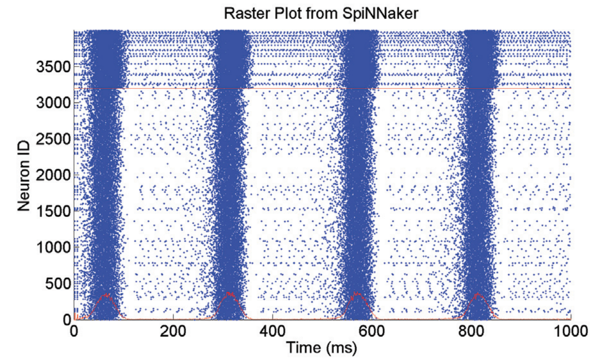
The timer on the Fascicle Processor is started after all other initialization processes (in both Fascicle and Monitor processors) complete, since the timer interrupt triggers the neuron state update, which requires all the data to be loaded. To avoid confliction, a handshake procedure is introduced to synchronize between the monitor processor and fascicle processor(s).



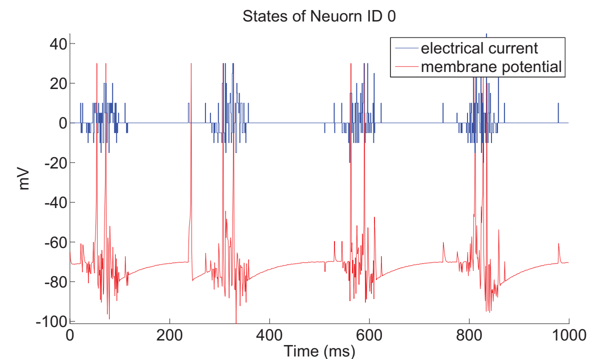
(a) 4000-neuron floating-point Matlab simulation.



(b) 4000-neuron fixed-point arithmetic Matlab simulation.



(c) 4000-neuron SpiNNaker simulation.



(d) States of neuron 0 (excitatory)

Fig. 10. Spike raster of 4000 neurons on SpiNNaker.

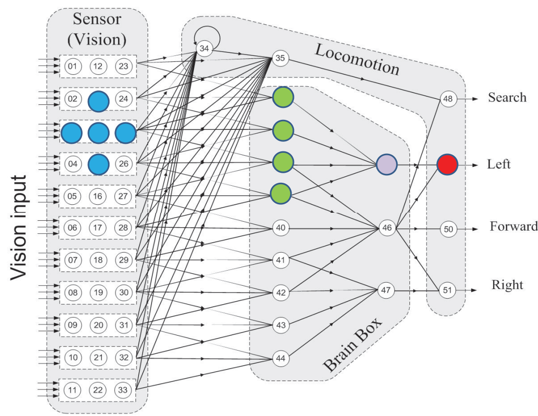


Fig. 11. The Doughnut Hunter model drawn by Arash Ahmadi.

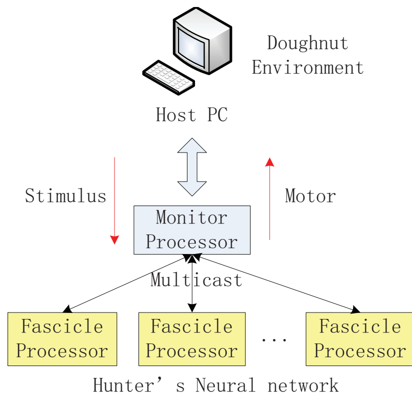


Fig. 12. The platform for the Doughnut Hunter application.

B. Simulation result

The test involves a simulation of a 4,000-neuron network (1,000 neurons per fascicle limited by the processing power [12]) with an excitatory-inhibitory ratio at 4:1. Each neuron is randomly connected to 26 other neurons. 72 excitatory and 18 inhibitory neurons are randomly chosen as biased neurons, each receiving a constant input stimulus of 20 mV. The simulation results are compared between the floating-point arithmetic Matlab simulation (Figure 10(a)), fixed-point arithmetic Matlab simulation (Figure 10(b)), and 4-chip SoC Designer based SpiNNaker simulation (Figure 10(c)). The spike timings in the floating-point and fixed-point arithmetic Matlab simulations are different, however, they show the same rhythm of 4Hz. The 4-chip SpiNNaker simulation matches the fixed-point arithmetic Matlab simulation. Figure 10(d) shows the activity of an excitatory neuron (ID 0) produced in the SpiNNaker simulation.

Each processor in a SpiNNaker chip is modeling 1,000 neurons, indicating that to model a human brain with 100 billion (10^{11}) neurons, at least 5 million full SpiNNaker chips with 20 processors per chip will be required.

C. The Doughnut Hunter on a real SpiNNaker chip

The Doughnut Hunter application was originally developed by Arash Ahmadi at Southampton and was ported onto SpiNNaker for testing. The model requires two application components, as shown in Figure 12: a server on the Host PC and a client on the SpiNNaker:

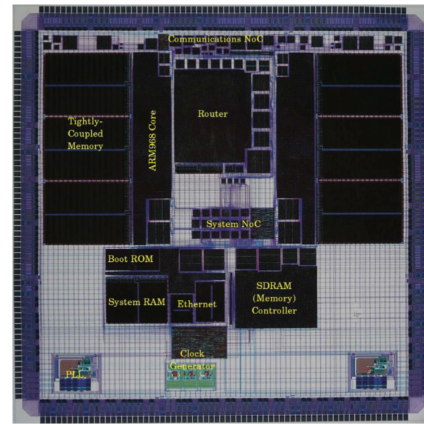


Fig. 13. The test chip diagram.

1) The server runs a GUI application modeling the environment with a doughnut and a hunter. The doughnut appears at a random location on the screen, and the hunter moves on the screen to chase the doughnut.The server detects the location of the doughnut and of the hunter, and provides the visual stimuli to the hunter at a certain frequency. These signals will be converted and sent to the client as input stimuli.
2) The client application runs on the SpiNNaker chip and models the neural network of the hunter: visual inputs is received from the server, and is propagated to the motor neurons through a simple neural network (as shown in Figure 11). The motor signal will be sent back to the server, used to control the hunter movement and update the hunter position. The connection between the server and client is via the Ethernet interface.

The Hunter neuron network is composed by 51 Fast Spiking Izhikevich Neurons, connected as shown in Figure 11; the application was successfully tested on the SpiNNaker Test Chip (two processors per chip) released in December 2009 as shown in Figure 13.

VII. Discussion

A. Real-time

SpiNNaker aims to simulate spiking neural networks in real-time with 1ms resolution. As a result, all the tasks must complete before the deadline – a new millisecond Timer Interrupt. Our assumption is the system should not be fully loaded to avoid losing real-time performance. This can be achieved by tuning the number of neurons or connections simulated on each processor.

In the scheduler we proposed as shown in Figure 5, a hard real-time [1] scheme is used, where the Timer INT is set to the

[1]The completion of an operation after its deadline in a hard real-time system is considered useless, and may ultimately cause a critical failure of the complete system.

second priority (or even to the first priority) and is very slim; this ensures that the Timer Interrupt request is responded to as soon as possible to guarantee the correctness of system timing. However, if an Timer Interrupt comes before the completion of all tasks, the processor is forced to move on to the new millisecond with the remaining tasks discarded, which leads to precision lost. Potential improvements can be made by using some algorithm to deal with the un-handled tasks. For instance:

1) Instead of simply clearing the packets in the communication buffer, they can be pushed into a history buffer associated with an old time stamp, and continue to be processed during the new millisecond with deferred timing.
2) Instead of skipping un-updated neurons, they can be updated with a reduced time resolution of 2 ms (so only 1 update is performed every 2ms), or with some simplified dynamics.

B. Neuron-processor mapping

The allocation of neurons to processors is an essential problem which affects the efficiency of both the communication and the synaptic weight storage. This problem, however, deserves further investigation. In principle, locality issues have to be taken into consideration to allocate relatively highly-connected nearby neurons onto the same or nearby processors as much as possible. There are three aspects that need to be taken into consideration for the neuron-processor mapping:

- Chip index. SpiNNaker chip IDs are purely software defined. The physical distance information needs to be included in the index for a well-defined mapping model. Each SpiNNaker chip contains up to 20 processors. The index of processors in the same chip does not matter since it takes the same time for them to communication to other chips.
- Neuron index. Neuron IDs are usually allocated randomly when a neural network is built, ignoring their "distances". Here the "distance" denotes the biological distance – how far a pair of neurons are apart from each other. A long axonal delay usually implies a long distance. For a well-defined neuron-processor mapping model, the "connectivity" and "distance" need to be considered when neurons are indexed.
- Building relations between the chip IDs and the neuron IDs. The principle is that neurons nearby (at a short distance) or with high connectivity need to be mapped on to the same or neighboring SpiNNaker chips.

The development of the neuron-processor mapping model for spiking neural networks relies on knowledge of a variety of spiking neural networks, and a well-defined mathematic model of mapping.

C. Load balance and synchronization problem

Because the clock domain of each chip is decoupled and each chip is expected to run in real-time, the SpiNNaker system doesn't have a synchronization problem. However, this requires a uniform distribution of workloads to guarantee that even the heaviest loaded processor won't lose real-time performance. Otherwise it cause precision lost as discussed above.

D. Code and data downloading

On a physical SpiNNaker machine, the downloading process will become a major issue involved in the Boot-up Process [13]. One difficulty is caused by the limited bandwidth between the SpiNNaker system and the Host PC, as Chip (0,0) is the only chip connected to the Host PC via Ethernet. Other chips will rely on their neighboring chips to receive data. To address this problem, a flood-fill mechanism is proposed where during the inter-chip flood-fill process, each chip sends one 32-bit word at a time by Nearest Neighbor packets to its 6 neighboring chips. The receiving chips then forward the data to their neighbors, so in this way, the data is propagated to all the chips in the system.

The application code is identical for every chip and processor, and therefore can be handled efficiently by the flood-fill mechanism. The neural data, on the other hand, is different for each chip or processor. The size of the data also increases when the system is scaled up. For a small system the flood-fill mechanism is capable to download neural data produced by the InitLoad software. For a large system, however, the huge amount of neural data results in an inefficient process. An alternative on-line data generation scheme is therefore required to send only statistical information of a neural network by the flood-fill mechanism. Each processor generates the real neural data as well as routing tables on-line according to the statistical information. This involves another potential research topic requiring a number of related issues to be solved, including how to produce the statistical information for a given neural network, how to modify routing tables dynamically, how to generate the synaptic weights for the event-address mapping scheme, and so on.

E. Data dumping

It is probably not feasible to dump the state of every neuron from the human brain. Usually, the state of a single neuron is not of direct interest, but the global output (for example the human language or behaviors) of the brain is of concern. If neuronal behaviors in a certain brain region are of interest during neuronal study, they can be observed using techniques such as glass electrodes, MRI or PET.

The SpiNNaker system is somewhat like the human brain in this respect. It is difficult to observe the activity of all neurons in the system because of the limited bandwidth between the Host PC and the SpiNNaker connection and the huge data flow. Users are still allowed to view the states of certain number of neurons in a certain time span however. One possible way to do this is for a user to send an observation request from the Host PC, which will be passed to the destination chips via the Host Chip (0,0). When the request is received by the Monitor Processor on the destination chip, it gathers the required information from

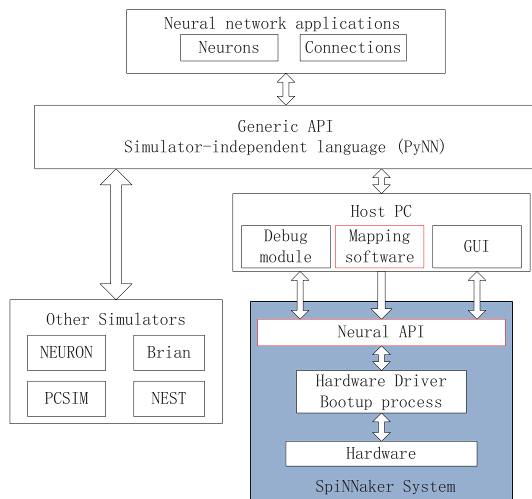


Fig. 14. The software architecture for SpiNNaker

local Fascicle Processors, and transmits the information to the Host PC, again via the Host Chip.

F. Software architecture for SpiNNaker

A good system requires not only high performance hardware but also an easy-to-use software system. Target users of the SpiNNaker system may not have sufficient time or skill to program such a complicated system. It will be more efficient to provide a system which is ready to use, without requiring the neural model to be changed.

Based on previous study, a software model is proposed as shown in Figure 14. the neural network model lies at the top-level where it is parsed by a generic API, PyNN for instance, which is a simulator-independent language for the neural network model. The generic API supports a variety of neural simulators such as NEURON, Brain, PCSIM, NEST, and SpiNNaker. This allows a user to write code for a model once, and run it on different models.

The Host PC provides mid-layer support to the generic API obtaining output information from the generic API, and converting the information to the format which can be loaded by the SpiNNaker system. The Host PC is also responsible for providing debugging facilities and GUI support to help users monitor system behavior. The SpiNNaker system lies at the bottom layer, shown in the blue square. From the top down, the SpiNNaker system comprises a neural API layer, a hardware driver layer, a bootup process, and hardware. The neural API layer includes libraries used to maintain the neural network model on SpiNNaker, such as updating neuron states, processing spike events, modeling synaptic plasticities and so on. The neural API layer requires low level support from hardware drivers. The bootup process is responsible for booting the system, checking the status of components, downloading application codes and so on.

By using this or similar software architecture, the hardware details of the SpiNNaker system can be hidden in a black box for end users, which makes it easier for users to run their code on a SpiNNaker system and to observe the results. Of course, there is much work required to fulfill this objective. The work presented so far mainly concerns the mapping software development, and the neural API implementation. Aspects of the hardware drivers and the bootup process development were also involved.

VIII. Conclusion

Modeling large scale neural networks on dedicated hardware opens new challenges. This paper made up a minimum system for testing and verifying the SpiNNaker system when simulating spiking neural networks. Potential issues found through this work are highlighted and discussed, leading the further development of SpiNNaker system towards simulation of larger scale networks with more complicated and meaningful neural models, on more chips.

Acknowledgment

We would like to thank the Engineering and Physical Sciences Research Council (EPSRC), Silistix, and ARM for support of this research.

References

[1] H. Markram, “The blue brain project,” *Nat Rev Neurosci.*, vol. 7, pp. 153–160, 2006.

[2] E. M. Izhikevich and G. M. Edelman, “Large-scale model of mammalian thalamocortical systems,” *PNAS*, vol. 105, pp. 3593–3598, 2008.

[3] L. Maguire, T. McGinnity, B. Glackin, A. Ghani, A. Belatreche, and J. Harkin, “Challenges for large-scale implementations of spiking neural networks on fpgas,” *Neurocomputing*, vol. 71, no. 1-3, pp. 13 – 29, 2007.

[4] M. Glover, A. Hamilton, and L. S. Smith, “An analog vlsi integrate-and-fire neural network for sound segmentation,” in *Seventh International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems*, 2009.

[5] P. A. Merolla, J. V. Arthur, B. E. Shi, and K. A. Boahen, “Expandable networks for neuromorphic chips,” in *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMSł*, February 2007.

[6] S. Furber and A. Brown, “Biologically-inspired massively-parallel architectures - computing beyond a million processors,” in *Proc. ACSD’09*, 2009.

[7] A. Rast, X. Jin, F. Galluppi, L. Plana, C. Patterson, and S. Furber, “Scalable event-driven native parallel processing: The spinnaker neuromimetic system,” in *ACM International Conference on Computing Frontiers 2010*, 2010.

[8] A. Rast, S. Yang, M.Khan, and S. Furber, “Virtual synaptic interconnect using an asynchronous network-on-chip,” in *Proc. 2008 International Joint Conference on Neural Networks*, HongKong, 2008, inproceedings.

[9] M. Khan, E. Painkras, X. Jin, L. Plana, J. Woods, and S. Furber, “System level modelling for spinnaker cmp system,” in *Proc. 1st International Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO’09)*, 2009.

[10] X. Jin, A. Rast, F. Galluppi, S. Davies, and S. Furber, “Implementing spike-timing-dependent plasticity on spinnaker neuromorphic hardware,” in *Proc. 2010 International Joint Conference on Neural Networks*, 2010.

[11] E. M. Izhikevich, “Which model to use for cortical spiking neurons,” *IEEE Trans. Neural Networks*, vol. 15, no. 5, pp. 1063– 1070, 2004, article.

[12] X. Jin, S. Furber, and J. Woods, “Efficient modelling of spiking neural networks on a scalable chip multiprocessor,” in *Proc. 2008 International Joint Conference on Neural Networks*, Hong Kong, 2008, inproceedings.

[13] M. M. Khan, “Configuring a massively parallel cmp system for real-time neural applications,” Ph.D. dissertation, Computer Science, University of Manchester, 2009.