Efficient Modelling of Spiking Neural networks on a Scalable Chip Multiprocessor

Xin Jin, Steve B.Furber, and John V.Woods

Abstract—we propose a system based on the Izhikevich model running on a scalable chip multiprocessor — SpiNNaker – for large-scale spiking neural network simulation. The design takes into account the requirements for processing, storage, and communication which are essential to the efficient modelling of spiking neural networks. To gain a speedup of the processing as well as saving storage space, the Izhikevich model is implemented in 16-bit fixed-point arithmetic. An approach based on using two scaling factors is developed, making the precision comparable to the original. With the two scaling factors scheme, all of the firing patterns by the original model can be reproduced with a much faster execution speed. To reduce the communication overhead, rather than sending synaptic weights on communicating, we only send out event packets to indicate the neuron firings while holding the synaptic weights in the memory of the post-synaptic neurons, which is so-called event-driven algorithm. The communication based on event packets can be handled efficiently by the multicast system supported by the SpiNNaker machine. We also describe a system level model for spiking neural network simulation based on the schemes above. The model has been functionally verified and experimental results are included. An analysis of the performance of the whole system is presented at the end of the paper.

I. INTRODUCTION

NEURONS and their connections are basic components of biological neural networks. A typical neural network comprises millions of neurons and billions of connections. Human brains have about 10^{12} neurons with each neuron connecting to thousands of others. To simulate a neural network at the scale of human brains is still unachievable; however, several systems are under construction to simulate large-scale neural networks [1, 2].

Because of the parallel nature of neural networks, a recent trend in computational neural science is to use parallel machines to simulate neural networks [3-5]. In a parallel system, processing and information storage are distributed and the system is usually more complex than a serial system. Such systems benefit from the multiple processing resources, but also face the drawbacks of communication overheads since information exchange is always required between the processing units and, to use a parallel system to simulate neural networks, one needs maximum use of parallel processing with minimum impact of the communication overheads. In this context, optimal processing of the neural model and efficient mapping of the neural network are

essential.

SpiNNaker is a scalable chip multiprocessor system designed for real-time large-scale spiking neural network simulation [6, 7]. Each SpiNNaker chip contains 20 ARM968E-S processors. Chips are organized in a *hexagonal* mesh, each communicating with 6 neighbors through asynchronous links. A multicast system is developed to make available efficient one-to-many communications to support the way neurons interact. We propose to use the Izhikevich model of spiking neurons in SpiNNaker for its simplicity in computation and richness in reproducing diverse neural firing patterns.

Three factors crucial to the efficient simulation of spiking neural networks are processing, storage, and communication. In this paper, we propose a scheme for the efficient simulation of spiking neural networks based on the Izhikevich model on the SpiNNaker system taking these three factors into account. Firstly, a 16-bit fixed-point implementation of the Izhikevich model is investigated to get a better processing speed as well as to save storage space. By adopting two scaling factors, the precision of the 16-bit implementation is still comparable to the original (floating-point) model. The architecture of efficiently mapping spiking neural networks onto the SpiNNaker machine is also presented, focusing on the communication efficiency and system scalability. By holding synaptic weights in the memory of post-synaptic neurons, the communication throughput required is reduced when using the multicast system supported by SpiNNaker.

The design is functionally verified and evaluated on a system modelled using the RealView ARMulator Instruction Set Simulator (RVISS).

II. NEURON MODELS

In biological neural networks, neurons communicate through spikes and use the timing of the spike to compute. Previous models, including the Mcculloch-Pitts model [8], the PDP model [5] and some others, are non-spiking since they do not employ individual pulses as do biological neurons. In recent years, models of spiking neurons have become increasingly popular as they capture the spiking nature of real neurons and can reproduce various neuron spiking patterns. Spiking neural networks based on these models are more biologically plausible and more powerful than non-spiking ones [9, 10].

The Hodgkin-Huxley model [11] is one of the most detailed and best known models of spiking neurons. It quantitatively describes the subcellular level ionic behaviors and the membrane current underlying the generation and propagation of neural spikes. Although it is biologically meaningful and powerful, the Hodgkin-Huxley model is too

computationally complex to use in large-scale neural network simulation. Only computationally simple models are practical in large-scale modelling because of the limited hardware resources. Some simplified models, such as the integrate-and-fire model, have been proposed as a consequence. These models can simulate the spiking nature of neurons as well as most key behaviours, while keeping the cost of the computation at a comparatively low level.

The Izhikevich model is a simplified model that has a good performance both on computational efficiency and functional richness [12]. The Izhikevich model is based on a pair of coupled differential equations:

$$v' = 0.04v^2 + 5v + 140 + I - u \qquad (1)$$

$$u' = a(bv - u) \qquad (2)$$

When $V \geqslant 30$,

$$v = c; \quad u = u + d \qquad (3)$$

Where $` = d/dt$, t is time (in ms), I is the synaptic current, v represents the membrane potential of the neuron (in mV), u represents a membrane recovery variable (also in mV). a, b, c, and d are parameters:

- the parameter a is the time scale of the recovery variable u. Smaller values results in slower recovery.
- b describes the sensitivity of the recovery variable u to the subthreshold fluctuations of the membrane potential v. Greater values couple v and u more strongly, resulting in possible subthreshold oscillations and low-threshold spiking dynamics.
- c describes the after-spike reset value of the membrane potential v.
- d describes the after-spike offset of the recovery variable u.

Notice that the threshold value of this model is between about -70 mV and -50 mV and is dynamic [13]. In this model, when the membrane potential v exceeds the threshold value, the neuron spikes with a 30 mV apex of membrane potential v. The membrane potential v is limited to 30 mV. If the membrane potential v goes above the limitation, it is firstly reset to 30 mV. Then the membrane potential v and the recovery variable u are both reset according to (3).

With the choice of neuron parameters a, b, c, and d, the model can exhibit different firing patterns. The Izhikevich model is so computationally simple compared to the Hodgkin-Huxley model that it takes only 13 floating-point operations to simulate 1 ms of the model, but can reproduce firing patterns of all known types of cortical neuron [13]. The H-H model takes 1200 floating-point operations for 1 ms.

III. SPINNAKER

The SpiNNaker project is a joint project between the Advanced Processor Technologies (APT) Group at the University of Manchester and the School of Electronics and Computer Science (ECS) at the University of Southampton. The objective is to build a scalable chip multiprocessor for real-time large-scale spiking neural network simulation.

The basic components providing the computational capability for the SpiNNaker system are the ARM968E-S processing subsystems, each of which is called a fascicle

processor and is responsible for modelling a number (in the region of 1,000) of neurons. Each subsystem comprises a 200 MHz ARM968E-S core with a 32 KB Instruction Tightly Coupled Memory (ITCM) and a 64 KB Data Tightly Coupled Memory (DTCM).

Each SpiNNaker chip contains 20 identical ARM968E-S processing subsystems, as shown in Fig.1, with a block of external SDRAM associated with each chip. Processors share access to the SDRAM through a self-timed packet-switched System NoC. CHAIN technology[14] is used in this fabric to provide a high bandwidth of around 1 GB/s [7]. There is a Multicast Router on the chip for one-to-many routing between chips and processors.
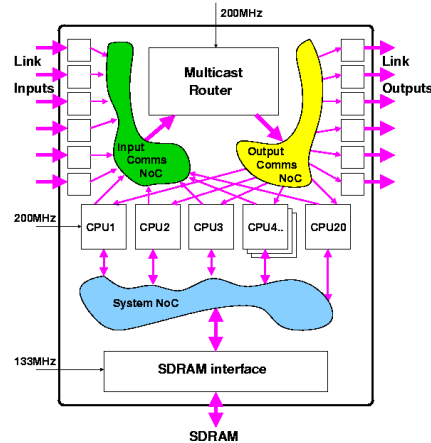

Fig.1. A SpiNNaker chip. Each chip contains 20 processors sharing an external memory block through the System NoC. Processors connect to the Multicast Router through the Communication NoC. There are 6 input ports and 6 output ports on each chip for inter-chip communication.

A SpiNNaker chip, together with its associated SDRAM, forms one processing node of the SpiNNaker system. A typical SpiNNaker system comprises a large number of processing nodes which are organized in a hexagonal 2D mesh, each communicating with 6 neighbours through self-timed packet-switched links. This kind of asynchronous connection decouples the different clock domains between and within SpiNNaker chips and, as a result, makes the system scalable. To meet the communication throughput demand of the high connectivity of the neural network system, 8-wire inter-chip link are used to obtain a capacity of around 1 Gbit/s when connecting two SpiNNaker chips [7, 15].

IV. 16-BIT FIXED-POINT IMPLEMENTATION OF THE IZHIKEVICH MODEL

The Izhikevich model of spiking neurons is selected as the neuronal model used in the SpiNNaker system. Floating-point numbers are used in the original Izhikevich model, however, fixed-point operations are normally more efficient than their floating-point alternatives and the ARM968E-S processor does not have a Floating Point Unit (FPU). As a result, we propose to use fixed-point operations instead of floating-point operations in the system. 16-bit fixed-point arithmetic is used to further speed-up the

processing and save storage space. Pearson *et al.* have implemented spiking neural networks based on fixed-point leaky-integrate-and-fire (LIF) model on FPGA [16, 17]. We are modeling spiking neural networks of a different neuron model on different hardware with a new approach of using two scaling factors.

A. Choice of scaling factors

To approximate the floating-point arithmetic by the fixed-point arithmetic we need to adopt scaling factors. The choice of scaling factors is essential in the floating-point to fixed-point transformation. To choose a proper scaling factor, firstly we investigate the ranges of the variables and parameters relevant to the transformation.

According to the experimental results from the simulation of the Izhikevich model, the value of the membrane potential *v during* computing is in the range -80 to 380, where 380 is the value before reset (It is reset to 30 immediately after it reaches 380 and then reset to *c*). A 16-bit half word can represent a signed integer number in the range -32768 to 32768. Hence we get

$$-32768 \le vp \le 32767 \quad (-80 \le v \le 380) \qquad (4)$$

Where, p is the scaling factor.

According to (4), we get $p < 86$.

In this case, we only consider values for p that are powers of 2 so that they can be implemented simply by shifting. Since a greater value of p always leads to better precision (see experimental results in Table I and Table II below), we choose $p = 64$. If we select any value for p greater than 64, the membrane potential v may overflow during computation.

However, ARM968E-S is a 32-bit processor. Some 32-bit operations are therefore as efficient as some 16-bit operations. This allows us to expand some operations from 16-bit to 32-bit *during* computation to gain better numerical precision without losing performance, and we can still keep variables in the data structure in 16-bit format. In this way, a greater value of p can be applied to produce better precision without increasing the computation time and the storage space.

Although the value of the membrane potential *v during* computing is in the range -80 to 380 as we described above, the final value of the membrane potential v hold in the data structure will be in the range -80 to 30. We get

$$-32768 \le vp \le 32767 \quad (-80 \le v \le 30) \qquad (5)$$

$p = 256$ can be selected to satisfy (5) .

So far, we have considered only the variable which has the greatest numerical value. Some parameters in (1) and (2) with very small floating-point values also need to be considered since they may cause a decrease of the precision if the value of the scaling factor is not big enough. There are two parameters a, b, and one constant 0.04 that we should care about in (1). a and b range roughly from 0.02 to 0.1 and from 0.2 to 0.25 respectively when modelling different types of neuron. The scaling factor $p = 256$ is probably just enough for these values. However, to get a better performance, some changes to the presentation of (1) and (2) are made (detailed below), in which case, parameters a and b are integrated into one parameter ab. ab is in the range 0.004 to 0.025, which

makes the precision worse when the equations are transformed to fixed-point using the scaling factor $p = 256$.

The solution we have adopted here is to use two scaling factors p_1 and p_2 with a small and large value respectively. We apply the smaller scaling factor p_1 to parameters, variables and constants with values greater than 0.5 and the larger scaling factor p_2 to those with values less than 0.5. $p_2 = 65536$ is selected because it is both large enough and efficient to implement using multiply-accumulate operations (detailed below). So we get

$$p_1 = 256, \quad p_2 = 65536 \qquad (6)$$

B. The transformation of the equations

In order to get an extremely fast processing speed, a few changes are made to the presentation of (1) and (2). These changes are made based on two objectives:

- Pre-computing as much as possible.
- Reducing the number of operations as much as possible.

Continuous-time differential equations (1) and (2) can be implemented in discrete-time by the following equations,

$$v = v + \tau(0.04v^2 + 5v + 140 + I - u) \qquad (7)$$

$$u = u + \tau a(bv - u) \qquad (8)$$

Where τ is time step which can be small to achieve adequate numerical precision. We set $\tau = 1$ in this paper for 1 *ms* resolution.

In the ARMv5TE architecture, there is a signed multiply-accumulate operation ($32 \le 32 \times 16 + 32$) -- "SMLAWB", where "B" means use the bottom half of the register (bits [15:0]). An operation with the form of $(ax \cdot b)/x + c$ can be implemented by one "SMLAWB" instruction when $x = 2^{16}$ and b is a 16-bit value. It takes only one CPU cycle to obey this instruction in the ARM968E-S.

We transform (7) and (8) to the following:

$$v = v(0.04v + 6) + 140 + I - u \qquad (9)$$

$$u = -au + u + abv \qquad (10)$$

After applying scaling factors p_1 and p_2, (9) and (10) turn out to be:

$$vp_1 = \{vp_1[(0.04p_2 \cdot vp_1)/p_2 \\ + 6p_1]\}/p_1 + 140p_1 + Ip_1 - up_1 \qquad (11)$$

$$up_1 = [(-ap_2) \cdot up_1]/p_2 \\ + up_1 + [(abp_2) \cdot vp_1]/p_2 \qquad (12)$$

We setup a new data structure for each neuron:

```
struct NeuronState
  {
    signed short Param_v;      // vp1
    signed short Param_u;      // up1
    signed short Param_a;      // abp2
    signed short Param_b;      // -ap2
    signed short Param_c;      // cp1
```

```
        signed short Param_d;        // dp₁
    }NeuronStates;
```

In this data structure, scaling factors p_1 and p_2 have been applied. Hence variables and parameters are fixed-point numbers and pre-computed. Constants $6p_1$ and $140p_1$ in (11) are also pre-computed. Meanwhile, neuron parameters are re-defined according to the new equations, i.e. neuron parameters a and b in (2) are replaced by ab and $-a$ respectively. In ARM assembly code, when $p_2 = 2^{16}$ (65536), 1ms simulation of (11) and (12) consists of:

1) $A = (0.04 p_2 \cdot vp_1)/p_2 + 6p_1$,   one   "SMLAWB" operation.

2) $A = A << (16 - \log_2 p_1)$, one shifting operation.

3) $A = \{A \cdot vp_1\}/p_1 + 140 p_1$, one "SMLAWB" operation.

4) $A = A + Ip_1$, one "ADD" operation.

5) $vp_1 = A - up_1$, one "SUB" operation.

6) $A = [(-ap_2) \cdot up_1]/p_2 + up_1$, one "SMLATT" operation, which is a signed multiply-accumulate operation (32 <= 16 x 16 + 32), "T" means use the top half of the register (bits [31:16]).

7) $A = A >> \log_2 p_2$, one shifting operation

8) $up_1 = A + [(abp_2) \cdot vp_1]/p_2$, one "SMLAWB" operation

Where A represents the partial result of each step. In step 1, vp_1 is stored in the bottom 16 bits of a register. When the "SMLAWB" instruction is obeyed, what it actually does is multiplying $0.04 p_2$ (32 bits) by vp_1 (16 bits) in the bottom 16 bits of a register and only the top 32 bits of the multiplication result is prereserved. If $p_2 = 2^{16}$, the division operation of "$/p_2$" is done automatically as the bottom 16 bits is dismissed during the multiplication. The result is added to $6p_1$ finally. However, in step 3, $p_1 \neq 2^{16}$, so we need a shift operation in step 2 to fit the condition. In step 6, $-ap_2$ and $-up_1$ is kept in the top 16 bits of two different registers respectively. The computational result of step 6 is in the most significant 16 bits. As a result, a shift operation is required in step 7.

In this approach, 1 ms simulation only takes 6 fixed-point mathematical operations plus 2 shifting operations. Obviously it is more efficient than the original which took 13 floating-point operations.

C. The precision

Different choices of scaling factors p_1 and p_2 lead to different levels of precision. Table I illustrates a comparison of the number of spikes generated in a certain period of time by different combinations of p_1 and p_2. Results generated from floating-point implementations of (7) and (8) with $\tau = 1$ are also given in Table I as benchmarks. The top sub-table in Table I comprises results from "tonic spiking" while the bottom sub-table comprises results from "tonic bursting". As we can see, greater values of the scaling parameters lead to better precision. Results from the simulation with $p_1 = 256$, $p_2 = 65536$ are very close to the benchmarks.

TABLE I
SPIKE COUNTS FOR TONIC SPIKING AND BURSTING

Tonic Spiking. Simulated for 20000 *ms*, 1*ms* resolution
a = 0.02, b = 0.2 c = -65, d = 6;
v(0) = -70, u(0) = 0.2v(0), I = 14 after 0 *ms*
Number of spikes (floating-point): *642*

Spike P₂	16-bit fixed-point		
	P₁ = 64	P₁ = 256	P₁ = 8192
256	449	437	482
2048	542	542	566
8192	596	620	611
65536	631	*654*	651

Tonic Bursting. Simulated for 5000 *ms*, 1*ms* resolution
a = 0.02, b = 0.2 c = -50, d = 2; threshold = 3;
v(0) = -70, u(0) = 0.2v(0), I = 15 after 22 *ms*
Number of spikes (floating-point): *502*

Spike P₂	16-bit fixed-point		
	P₁ = 64	P₁ = 256	P₁ = 8192
256	364	375	393
2048	424	443	454
8192	449	444	495
65536	462	*501*	502

The above table shows a comparison of the number of spikes generated in a certain period of time by different choices of p_1 and p_2 in fixed-point simulation. Results from the floating-point implementation are also given as benchmarks. Results in the top sub-table are for "tonic spiking" while results in the bottom sub-table are for "tonic bursting".

Our approach can meet the requirement to reproduce all firing patterns with good precision as illustrated in Table II. In Table II, the number of spikes of four different patterns generated in a certain period of time by the fixed-point and the floating-point simulation respectively are compared. Other patterns which can be reproduced by the original model can also be modelled by this implementation. According to these results, the numbers of spikes generated by the fixed-point simulation are exactly the same as those generated by the floating-point simulation.

TABLE II
SPIKE COUNTS FROM DIFFERENT SPIKING PATTERNS

Simulated for 1000 *ms*, 1*ms* resolution P₁ = 256, P₂ = 65536				
Spikes	TS	TB	RS	IIS
Fixed-Point	34	102	1	6
Floating-Point	34	102	1	6

This table shows a comparison of the number of spikes generated in a certain period of time by the fixed-point simulation with the number of spikes generated by floating-point simulation. Results from 4 different firing patterns are listed. TS represents the "tonic spiking", TB represents the "tonic bursting", RS represents the "rebound spiking" and IIS represents the "inhibition induced spiking".

In addition to the number of spikes, we also evaluate the level of precision by other schemes. However, in some cases, the precision is not ideal. Table III shows the input current required to reproduce the pattern of rebound spiking with different choices of scaling factors. The result from the floating-point simulation is also given as the benchmark. If we choose $p_1 = 256$, $p_2 = 65536$, only when the input current $I = -50$ units can the rebound spike be reproduced, while in the

floating-point simulation, the required input current I is -21. If we choose p_1= 8192, p_2= 65536 or even greater values, then the result is comparable to the result from the floating-point simulation. However, this is not achievable in 16-bit fixed-point arithmetic.

According to the result, although floating-point to fixed-point transformation does have drawbacks in respect of the precision, it still can reproduce firing patterns with an acceptable level of precision.

TABLE III
INPUT CURRENT REQUIRED FOR REBOUND SPIKE

Rebound Spike. Simulated for 200 *ms*, 1*ms* resolution a=0.03, b=0.25, c=-60, d=4; v(0) = -64, u(0) = 0.2v(0), I lasts for 5 *ms* Floating-point: ***I = -21***				
I	16-bit fixed-point			
P_2	P_1 = 64	P_1 = 256	P_1 = 8192	P_1=65536
32768	-48	-50	-50	-50
65536	-49	***-50***	-23	-23

The input current I required to reproduce the pattern of "rebound spiking" with different choices of scaling factors is listed. The result from the floating-point simulation is given as the benchmark.

D. The processing speed

We program in assembly code using the RealView ARMulator Instruction Set Simulator (RVISS) 1.4 supplied with the RealView Developer Suite (RVDS) 2.2. RVISS simulates the instruction sets and architecture of ARM processors together with a memory system and peripherals. It can be used for software development and for benchmarking ARM architecture targeted software[18]. In RVISS, we model the system with a 200 *MHz* ARM968 core, a 100 *MHz* AHB bus, a 32 *KB* ITCM, a 64 *KB* DTCM and a 100 *MHz* SDRAM.

If we store data structures of neurons in the DTCM, 1 *ms* simulation of processing of (11) and (12) (i.e. an update of one neuron state in one millisecond), takes 240 *ns* if resetting the input current I to a constant number after updating and it takes 330 *ns* if resetting I to a random number (adding noise to the input). If we store the neuron data structures in the SDRAM, it takes 660 *ns* when resetting the input current I to a constant number.

More analysis of the processing speed is presented in chapter VI.

V. MAPPING SPIKING NEURAL NETWORKS

The SpiNNaker chip has efficient on-chip and inter-chip connections and a multicast mechanism for high-performance communication. Each fascicle processor in the system models a bunch of neurons. Neurons have to communicate to each other based on their connections. To map neural networks onto SpiNNaker, we have to solve the problem of how to distribute processing workloads while keeping the communication overhead low. This is a common problem in the parallel computing domain.

Generally speaking, there are two types of algorithm for the simulation of spiking neural networks: clock-driven and event-driven. In the former, all neurons are updated at every tick of the clock while in the latter, neurons are updated only when they receive or emit a spike [19]. The proposed algorithm is a combination of clock-driven and event-driven. Firstly, as in the clock-driven algorithm, neurons in our system are updated every millisecond as the differential equations are solved. Then, at the point when a spike (event) arrives, the value of the synaptic weight will be added to an element in a circular array in the neuron data structure where the input currents I are held. The circular buffer is indexed by time. The detail of the algorithm and the mapping strategy are described in the rest of this section.

A. Propagation of spikes

In a neural network system, when a neuron fires, it generates a spike that propagates to the post-synaptic neurons it connects to. Each connection is associated with a synaptic weight which indicates the strength of the effect that the pre-synaptic neuron has on the post-synaptic neuron. There is a high density of fan-out transmissions of dissimilar packets due to the high connectivity of neural networks. This pattern of traffic leads to inefficient communication on parallel hardware.

The heart of the communication system of the SpiNNaker chip is the Multicast Router mechanism. Using this mechanism, one-to-many communication with identical packets is efficient. As a result, we propose a strategy that when a neuron fires, it sends out an event packet as a spike to others only to indicate that it has fired. We use a routing key as the event packet which comprises the source fascicle ID and the neuron ID. The multicast router directs the routing key to one or more destinations based on its source fascicle ID and a route lookup table in each router. Finally, identical routing keys will arrive at each destination fascicle.

Synaptic weights are held at the post-synaptic ends. When the post-synaptic neuron receives a routing key, which indicates that one of its pre-synaptic neurons fired, it will look up its local memory to find out the synaptic weight associated with this connection based on the source fascicle ID and neuron ID in the routing key. More details about the routing key and the routing algorithm of SpiNNaker can be found in [20].

B. Storage of synaptic weights

The SDRAM is used for synaptic weight storage. As illustrated in Fig.2, synaptic weights are organized into banks in the SDRAM determined by the fascicle on the chip they belong to. Each fascicle owns one bank. Synaptic weights in the bank, indicated as "Wgt for LocFasc" in Fig.2, are only for the fascicle which possesses this bank of weights.

In each bank, synaptic weights are sorted into groups by source fascicles which have any pre-synaptic neuron connecting to post-synaptic neurons on this fascicle. Each group contains synaptic weights of all connections from neurons on a source fascicle to neurons on this fascicle, indicated as "Wgt for SrcFasc" in Fig.2. Each fascicle maintains a lookup table which can be used to look for the memory address of the group based on the source fascicle ID in the routing key received.

Each group again comprises several blocks. Each block contains synaptic weights for all connections from one

pre-synaptic neuron (on the corresponding source fascicle) to post-synaptic neurons on this fascicle, which is indicated by "Wgt for SrcNero" in Fig.2. A variable "*BlkSize*" is defined to indicate the number of connections. It is a scenario of one-to-many connections: one neuron on the source fascicle and many neurons on the local fascicle.
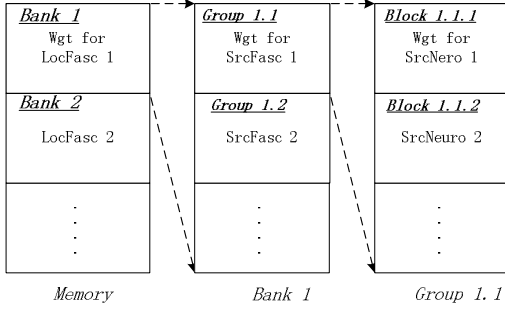

Fig.2. Synaptic weights storage. Synaptic weights are organized in different levels. Each bank comprises several groups and each group again comprises several blocks.

Each entry in the lookup table is a 16-byte data structure corresponding to a source fascicle. They are organized in a binary tree with all "<" branches occupying contiguous memory locations.

```
struct SFascicle
{
    int SFascAddr;              //address to match
    int SFascMask;              //bits to ignore
    int *SFascPtr;              //Sdram address of fasc start
    struct SFascicle *NextSFasc;        //go here if >
};
```
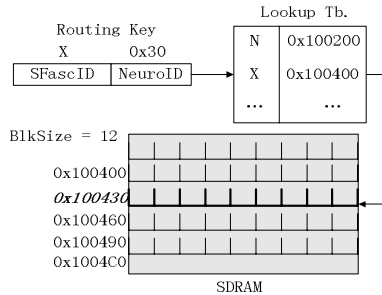

Fig.3. Lookup of the block of synaptic weights. The lookup table on the fascicle contains a list of source fascicle IDs each of which points to a base address of memory in the SDRAM. When a routing key arrives at the fascicle, the fascicle will search its lookup table based on the fascicle ID in the routing key. If it finds a hit, the base address to which the fascicle ID points (the group address) will be fetched. After that, the neuron ID in the routing key will be added to the base address to produce a full start address of the block. In this figure, the block size is 12 Words (48 *bytes*).

In the lookup table, each entry comprises a source fascicle ID and a memory address. The memory address points to the start address of the group associated to the source fascicle.

An example of locating a block of synaptic weights is illustrated in Fig.3. In Fig.3, a routing key arrives at the local fascicle with a source fascicle ID "X" and neuron ID 0x30, which indicates that the 0x30 neuron in the "X" source

fascicle has fired. The local fascicle then searches its lookup table for any entries matching the fascicle ID "X". In Fig.3, in terms of the source fascicle ID "X", we get the start address 0x100400 as a base address. The group of weights in the memory starting at address 0x100400 are for connections coming from the source fascicle to the local fascicle. Then, we need to find the right block of weights for the fired neuron based on the neuron ID. In this case, the block of weights starting at memory address 0x100430 is for the neuron fired. If the block size for one neuron is 12 (*BlkSize* = 12), this indicates that the fired neuron connects to 12 neurons in the local fascicle. All of the weights in this block have to be added into the circular arrays of the 12 respective neurons.

C. Synaptic Delay

Synaptic delays are the latency of the neuron communication. A spike generated by the pre-synaptic neuron may take time (in milliseconds) to arrive at the post-synaptic neuron. This is also considered in this algorithm. Each synaptic weight has a structure of

|--4b-delay-|0|-11b-index-|-16b-weight-|.

The highest 4 bits are for the synaptic delay and the lowest 16 bits are for the synaptic weight. The 11-bit index in the middle corresponds to the index of the post-synaptic neuron on the fascicle of this connection. The highest 4 bits allow us to simulate up to 16 *ms* synaptic delays.
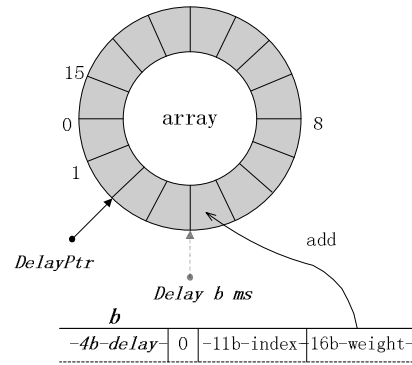

Fig.4. The circular array. A spike arrives after *b ms*, so the synaptic weight is added to the position of "*DelayPtr* + *b*" in the array.

Each data structure of the neuron possesses a circular array which contains 16 half-word integer numbers. Each element of the array corresponds to the input current *I* arrives at different time with 1*ms* resolution.

```
Neurons
{
    struct NeuronState
    signed short   Bin[16];          //delayed input
};
```

If a pre-synaptic neuron *i* spikes at time *a* ms, and a spike arrives at the post-synaptic neuron *j* after *b* ms, the synaptic weights of the connection from neuron *i* to neuron *j* will be added to the *I* in position "*a+b*" of the circular array belonging to the post-synaptic neuron. In the real-time simulation, *a* is always equal to the current time and there is a

pointer -- "*DelayPtr*" pointing to the current time in the circular array. As a result, "*a+b*" is equivalent to "*DelayPtr +b* "as shown in Fig.4.

D. System scheduling

In the implementation, to speed up the progress of adding synaptic weights to the circular arrays, before starting the process we move the whole block of weights for one fired neuron from the SDRAM to the DTCM by a DMA operation (the DTCM is much faster than the SDRAM), once the address of the block in the SDRAM has been calculated. As a result, the system is driven by three event signals:

1) The clock (every 1*ms*)
2) The arrival of spikes (packets)
3) The completion of the DMA operation.

A co-operative multi-tasking system is designed to schedule the event handling. There are three tasks corresponding to the three signals for each fascicle processor:

1) *"Update neuron state"*, which is executed every 1 *ms* to update the states of all neurons on the fascicle.
2) *"New input processing"*, which is invoked by the packet arrival signal to identify the fired neuron and work out the memory address of the synaptic weights based on the routing key. Then, start a DMA operation.
3) *"Update circular arrays"*, which is invoked by the DMA completion signal to update the circular arrays of the respective neurons.
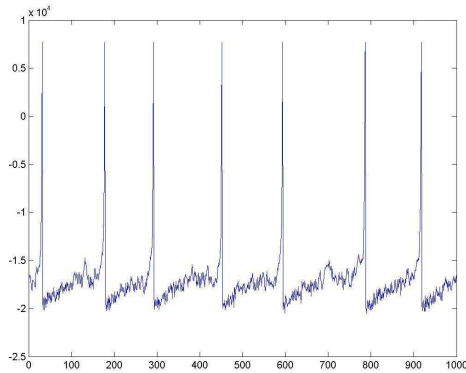
VI. SYSTEM PERFORMANCE

Fig.5. A firing pattern of one second simulation generated by one of neurons in the simulation

We simulate the behaviour of one fascicle processor based on the above algorithm again on RVISS. The RVISS model is the one described in section IV. To make the fascicle processor self-contained, we wire neurons on the same fascicle together. Spikes are sent to the buffer rather than through the real communication system, since only one fascicle is modelled. Neuron parameters are set in accordance with those in [13] to generate comparable results. 1000 neurons are modelled. Neurons are connected to each other randomly with a connectively around 10%, so each neuron has 100 inputs.

Fig.5 shows a firing pattern of one second simulation in

real-time generated by one of the neurons on the fascicle processor during the simulation.

A. Processing speed

The result of experiments shows that,

1) "Update neuron state". As shown above, it takes 240 *ns* to update one neuron state.
2) "New input processing". It takes 280 *ns* to process an incoming packet and start a DMA operation.
3) "Update circular arrays". It takes 110 *ns* to update one weight (4 *bytes*).

So the processing time of each fascicle can be estimated by *240 ns/Neuron + 280 ns/Input + 110 ns/Connection*

Where, in a certain period of time, *Neuron* is the number of neurons which have to be updated, *Input* is the number of incoming packets, *Connection* is the number of connections which have to be updated.

If a fascicle models *N* neurons with *I* inputs (incoming packets) each firing at *F Hz* and each neuron connects to all of others with a percentage of C, let *T* to be the processing time required (CPU time in *ns*) for 1 ms real-time simulation, we get

1) *N* neurons have to be updated per millisecond.
2) $F \cdot I / 1000C$ inputs arrive per millisecond.
3) $F \cdot I \cdot N / 1000$ connections have to be updated per millisecond.

For each fascicle, we get

$$T = 240N + 0.28FI / C + 0.11FIN \qquad (13)$$

According to (13), the processing time required *T* increases with the increase of the number of inputs per neuron *I* and the firing rate *F*. When keeping others constant, the inputs per neuron *I* and the firing rate allowed *F* for each neuron are mutual exclusive from the processing time point of view. Modelling 1000 neurons in real-time with 10% connectivity on one fascicle, it allows a firing rate up to 67 *Hz* on according to (13), which is quite close to the result got from the simulation which is 60 *Hz*.

B. Data memory usage

DTCM usage:

1) Neuron data structures. Each neuron data structure is 44 *bytes*, including two 16-bit variables, four 16-bit parameters, and a 16-bit circular array which takes 32 *bytes*. Modelling 1000 neurons takes *44 KB* of the DTCM space in total.
2) Lookup table. Each entry for one source fascicle is 16 *bytes*. Let the number of source fascicles in the lookup table be *M,* then requires *16M bytes.*
3) DMA buffer. The size of DMA buffer depends on the implementation. Each block of DMA has to be *4 bytes*BlkSize* which equals to *4NC* bytes. If a smaller size of DMA buffer is wanted, a bigger input buffer may be required. In our simulation, a double-buffer scheme is used, which takes *8NC* = 800 *bytes* in total.

SDRAM usage:

There are *NI* connections on one fascicle. Each connection takes 4 *bytes*. So 4*NI* bytes are required in total. Each SDRAM is shared by 20 fascicles; therefore one SpiNNaker

chip requires $80NI$ *bytes* if all processors on a chip are used for neuron modelling.

If each fascicle simulates 1,000 neurons, modelling 1,000 neurons with 100 inputs each (10% connectivity) requires 400 *KB* and uses only one processor. Modelling 20,000 neurons with 1,000 inputs each (5% connectivity) requires 80 *MB* in the SDRAM and uses one SpiNNaker chip. To model a large-scale neural network, we have to localize the connections to limit the SDRAM usage.

C. Communication

Components in SpiNNaker most related to the communication performance are the Multicast Routers and the links. Each router processes three types of packets categorized by their destinations: incoming packets, outgoing packets, and bypassing packets. The traffic and bandwidth required can be estimated based on these packets. Six links between one SpiNNaker chip and its 6 nearest neighbours support a total of 6 *Gbit/s* of bandwidth. In addition to the scale of the system modelled, the communication pattern on routers and links also relies on the routing scheme. Some description of their performance can be found in [7], [15] and [20]. More detailed analysis of the system throughput and communication patterns will be given in future.

VII. CONCLUSION

In this paper, we present an efficient way of modelling a spiking neural network on a scalable chip multiprocessor architecture. The design focuses on three aspects: the processing speed, the memory usage and the communication. Some exciting experimental results and analysis show that the system is capable of simulating large-scale neuron networks at 1*ms* resolution efficiently. Although the implementation proposed in this paper is hardware-specific, it still provides some generic ideas for the design of hardware platforms for computational neural networks.

ACKNOWLEDGMENT

REFERENCES

[1] K. Asanovic, J. Beck, T. Callahan, J. Feldman, B. S. Irissou, B. Kingsbury, P. Kohn, J. Lazzaro, N. Morgan, D. Stoutamire, and J. Wawrzynek, "CNS-1 Architecture Specification," EECS Department, UC Berkeley 1993.

[2] T. Schoenauer, N. Mehrtash, A. Jahnke, and H. Klar, "MASPINN: Novel Concepts for a NeuroAccelerator for Spiking Neural Networks," presented at Proc. VIDYNN'98, Stockholm, June 22-26, 1998.

[3] C. Wolff, G. Hartmann, and U. Rückert, "ParSPIKE A Parallel DSP-Accelerator for Dynamic Simulation of Large Spiking Neural Networks," presented at Proceedings of the 7th International Conference on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems, 1999.

[4] S. K. Foo, P. Saratchandran, and N. Sundarajan, "Parallel Implementation of Backpropagation Neural networks on a Heterogeneous Array of Transputers," *IEEE Transactions on Systems*, vol. 27, 1997.

[5] D. E. Rumelhart, J. L. McClelland, and t. P. R. Group, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 1: Foundations*: Cambridge, MA: MIT Press 1986.

[6] S. B. Furber, S. Temple, and A. D. Brown, "High-Performance Computing for Systems of Spiking Neurons," in *The AISB'06 workshop on GC5: Architecture of Brain and Mind*. Bristol, 2006.

[7] S. B. Furber, S. Temple, and A. D. Brown, "On-chip and Inter-Chip Networks for Modelling Large-Scale Neural Systems," in *Proc. ISCAS'06*. Kos: 2006, 2006.

[8] W. McCulloch and W. A. Pitts, "A logical calculus of the ideas immanent in nervous activity," in *Bulletin of Mathematical Biophysics*, vol. 5, 1943, pp. 115-133.

[9] W. Maass., "Noisy spiking neurons with temporal coding have more computational power than sigmoidal neurons," *Adavances in Neural Information Processing Systems*, vol. 9, 1997.

[10] W. Maass., "Networks of spiking neurons: the third generation of neural network models," *Neural Networks*, vol. 10, pp. 1659-1671, August 1997.

[11] A. L. Hodgkin and A. Huxley, " A quantitative description of membrane current and its application to conduction and excitation in nerve," *Physiol. London*, vol. 117, pp. 500–544, 1952.

[12] E. M. Izhikevich, "Simple Model of Spiking Neurons," *IEEE Trans. Neural Networks*, vol. 14, pp. 1569- 1572, Nov. 2003.

[13] E. M. Izhikevich, "Which Model to Use for Cortical Spiking Neurons," *IEEE Trans. Neural Networks*, vol. 15, pp. 1063- 1070, Sep. 2004.

[14] W.J.Bainbridge and S. B. Furber, "CHAIN: A Delay-Insensitive Chip Area Interconnect," presented at IEEE Micro, special issue on the Design and Test of System-on-Chip, 2002.

[15] A. D. Rast, S. Yang, M. Khan, and S. B. Furber, "Virtual Synaptic Interconnect Using an Asynchronous Network-on-Chip," presented at IJCNN2008, HongKong, 2008.

[16] M. J. Pearson, A. G. Pipe, B. Mitchinson, K. Gurney, C. Melhuish, I. Gilhespy, and M. Nibouche, "Implementing Spiking Neural Networks for Real-Time Signal-Processing and Control Applications: A Model-Validated FPGA Approach," presented at Neural Networks, IEEE Transactions on, 2007.

[17] M. Pearson, I. Gilhespy, K. Gurney, C. Melhuish, B. Mitchinson, M. Nibouche, and A. Pipe, "A Real-Time, FPGA Based, Biologically Plausible Neural Network Processor " *Artificial Neural Networks: Formal Models and Their Applications - ICANN 2005*, vol. 3697/2005, 2005.

[18] http://www.arm.com, "RealView ARMulator ISS Version 1.4 User Guide," 2004.

[19] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J. Bower, M. Diesmann, A. Morrison, and P. Goodman, "Simulation of networks of spiking neurons: a review of tools and strategies," *Comput Neurosci.*, vol. 23, pp. 349-398, 2007.

[20] D. R. Lester, M. M. Khan, S. B. Furber, L. A. Plana, A. Rast, X. Jin, and E. Painkras, "SpiNNaker: Mapping Neural Networks onto a Massively-Parallel Chip Multiprocessor," *IJCNN2008*, 2008.