Real-time Application Support for a Novel SoC Architecture

M.M. Khan*, J. Navaridas†, X. Jin*, L.A. Plana*, J.V Woods* and S.B. Furber*

***Abstract*— SpiNNaker is a novel SoC design with embedded low-power ARM processors. A system comprising multiple chips can provide a simulation engine for very large scale neural network simulations in real-time. The system has specifically been designed to support biologically inspired spiking neuron simulation as a real-time event-driven application. We have devised a novel technique to efficiently configure the system and support application development using an event driven application model by abstracting the architectural intricacies from the application developer. The main objective is to provide an efficient and fault-tolerant interface to the developers to make optimal use of the design features with minimal code size, meeting hard bounds of a real-time application.**

I. Introduction

THE SpiNNaker Massively Parallel Neural Networks Simulator is based on System-on-Chip (SoC) architecture to support parallel distributed computing for high performance and fault-tolerance. The whole SpiNNaker system comprises a network of smaller SoCs working as Chip-Multiprocessors (CMP) each consisting of 20 low-power ARM968S-E processing cores which can run neural dynamics models independently. The system has been designed to mimic neural computation [1] which is characterized by massive processing parallelism and a high degree of interconnection among the independently working processing units [2]. However, contrary to typical parallel computing models, the embedded processors do not rely on shared memory message passing to synchronize independently running processes. The system is highly scalable to support a neural simulation from thousands to millions of neurons with varying degree of connectivity. A full scale computing system is expected to simulate over a billion neurons in real-time employing over a million processing cores to simulate the behaviour of a part of neocortex. To support intense inter-neuron communication, a highly efficient asynchronous packet-switching routing network has been devised to connect the embedded processing cores. Figure 1 shows a logical view of the SpiNNaker computing system. The network is used for inter-process communication among the processing cores using small (40-bits) packets.

Inspired by the structure of the brain, the embedded ARM968 processing cores have been arranged in independently functional and identical Chip Multi-Processors (CMP) for fault-tolerent and scalable distributed computing. Each embedded core is self-sufficient in memory required to hold the neural modeling code and the neuron states of a number of neurons depending on the complexity of spiking neuron's model, while the chip has sufficient memory to hold synaptic information for the neurons in the system connected to the local neurons. The system at the chip- and system-level has sufficient redundancy in processing resources and communication infrastructure to provide a high level of fault-tolerance. The platform makes a large-scale neural simulation possible in real-time which would otherwise take days to simulate with a software simulation on a personal computer (PC). This real-time embedded system can be used as a 'brain' for a robot [3] to simulate real-time stimulus-response behaviour of a living being.

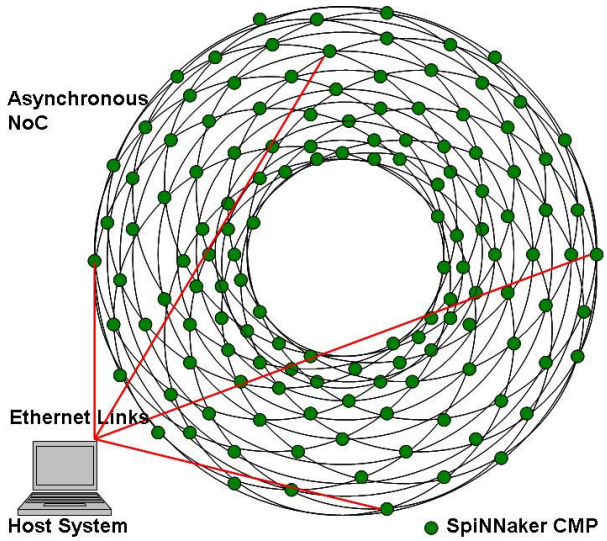

Fig. 1. Multichip SpiNNaker CMP System.

To support the standard application model of spiking neurons simulation on SpiNNaker an event-driven software model has been proposed that is quite different from a typical parallel application. To conserve energy and thus to reduce the heat dissipated from over a million processors, each embedded processor remains in sleep mode. The processors are woken up normally by neural events like the arrival of spike or the time when each neuron needs to update its state etc. The processor handles the event using a specific embedded service routine before going to sleep again. Unlike other parallel processing applications, no continuously running processes are created on the embedded cores. The architecture requires a novel technique to configure the system and to support a different kind of application model running on it. As part of this research, we have devised a novel mechanism to configure this system, load the application and support the running of spiking neural simulations using an event driven application model. Besides this, we provide a hardware abstraction layer to help the application develpers

The *authors are with the School of Computer Science, The University of Manchester, UK (email: khanm@cs.man.ac.uk). J. Navaridas† is with the University of The Basque Country, Spain.

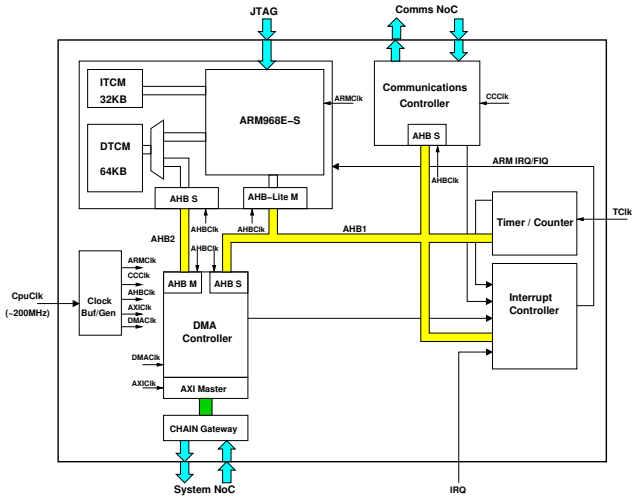

Fig. 2. Embedded ARM Core with Peripherals.

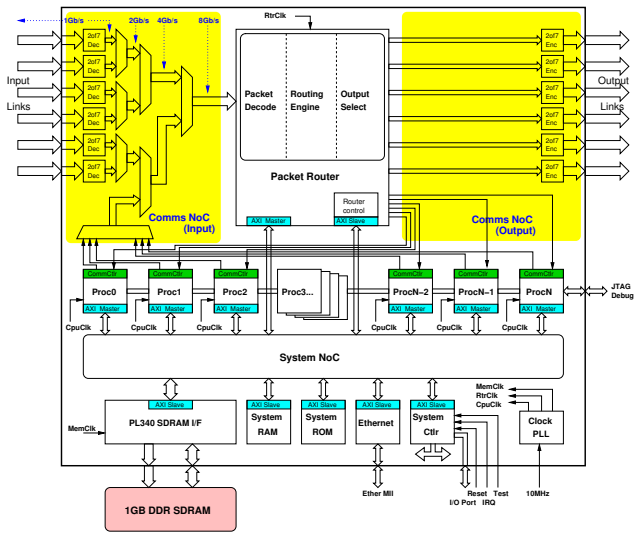

Fig. 3. SpiNNaker CMP.

making optimal use of the design features without knowing the architectural complexities of the system.

II. SpiNNaker Architecture

The SpiNNaker SoC has been designed to support highly parallel distributed computing with high bandwidth inter-processors communication. 20 ARM968 processing cores are embedded into each chip with a dedicated tightly-coupled memory that can hold 32KB of instructions and 64KB of data [4], sufficient to implement a fascicle (group of neurons with associated inputs and outputs) of about 1000 simple spiking neurons. Each embedded processor is provided with other peripherals such as a Timer, Vector Interrupt Controller (VIC), Communication and DMA Controller to support neural computation as shown in Figure 2. Besides the ARM cores' private memory, each chip also has an off-chip SDRAM of up to 1GB to hold connection information like synaptic weights and axonal delays. The off-chip memory module supports easy upgrading of memory a size as large as required by the application. The SDRAM is connected to the processing cores through the DMA Controller *via* an asynchronous Network-on-Chip (NoC), a high-bandwidth shared medium among the 20 embedded cores [5]. The network is called the System NoC and it provides a bandwidth of 1Gb/s. Other chip resources, like the System RAM, Boot ROM, System Controller and the Router Configuration Registers are also connected to the processing cores *via* the System NoC as shown in Figure 3. The asynchronous NoC provides a much higher communication bandwidth with lower contention and a very low power consumption [6][7] compared to any typical bus architecture.

To support event-driven modeling of spiking neurons as described in [8], the Timer generates an interrupt with a millisecond interval notifying the processing core to update each neuron's state [9]. Another event is generated by the Communication Controller on receipt of a spike in the form of a multicast packet from some other fascicle neuron. The Communication Controller is also responsible for forming a (40-bits) packet with source identifier (containing the firing neuron's identifier along with its fascicle identifier and chip address) on behalf of firing neurons in the fascicle. The spike transmission is carried over yet another asynchronous NoC called the Communications NoC. The hub of this NoC is a specially designed on-chip multicast router that can route packets (spikes) to 20 internal outputs corresponding to on-chip processing cores and 6 outward links towards other chips as a result of source-based associative routing. Each chip has links from its six neighbours that terminate in the Communications NoC where these, along with the internal packets, are serialized before passing to the router. The six two-way links on each chip extend the on-chip Communications NoC to the six neighbouring chips to form a system-wide global packet-switching-network. A system of the desired scale can be formed by networking chips with the help of these links, continuing this process until the system wraps itself around to form a toroidal mesh of interconnected chips as shown in Figure 1. The router contains 1K words of associative memory as a look-up table to find a multicast route associated with the incoming packet's routing key. If a match is not found, the router passes the packets towards the link diagonally opposite to the incoming link. This process is called 'default routing' and helps in reducing the number of entries in the look-up table. The global packet-switching network to support spike communication has been hierarchically organized. On the global network, only chip addresses are visible. Each chip maintains a chip-level private subnet of 20 fascicle processing cores that is visible only to the local router, while an individual neuron's identifier is local to the processing core. With this scheme we can manage to access all neurons with limited number of entries in the routing tables. To deal with transient congestion at the outer links,

the router can route a packet to its adjacent link as a measure of emergency routing. The neighbouring chip's router aligns the packet back to its correct path. The router is an efficient hardware component that can route one packet per cycle at 200 MHz. The Communications NoC supports up to 6 Gb/s per chip bandwidth [6]. The router can also route point-to-point(P2P) and nearest neighbour(NN) packets which are used for system management and debugging/configuration purposes. With an NN packet a chip can look into the state of its neighboring chip and write into its chip level memory. The router can broadcast an NN packet to all neighboring chips.

The SpiNNaker system is connected to a personal computer called the Host System by linking one (or more) chip(s) through an on-chip Ethernet link. The system is configured at boot-up and the routing tables are computed and loaded to each chip as per the application's setting with the help of the Host System. The application is injected to the connected chip(s) using Ethernet frames, from where it is flood-filled to other chips *via* the global asynchronous packet-switching network.

III. Application Model

SpiNNaker is an application specific integrated computing (ASIC) system that has been specifically designed to support large scale spiking neural network simulations in real-time (as suggested by the project name, Spi(king) N(eural) N(etwork) a(r)ch(it)e(ctu)r(e)). The architecture supports simulation of biologically plausible spiking neuron models with a reasonable computational complexity like that from Izhikevich [10]. Simulation results [9] show that a group of 1000 spiking neurons can easily be simulated on each embedded ARM968 processor with the help of its tightly coupled instruction and data memory. In line with biological neural state dynamics, the state of every neuron is affected by certain events such as the receipt of spikes. Each spike reaches the neuron from a particular dendritic connection with some synaptic weight associated with it. A weighted input from the spiking connection accumulates to the total stimulus to the neuron at a particular point in time. Biology suggests that the neurons update their state in millisecond temporal domain [11]. To realize this temporal notion, a timing event has been provided to the processing core by the Timer after every millisecond. Each neuron can have over 1000 synaptic connections to receive spikes from other neurons' axons. The spike transmission speed in axons is in the range of a few meters per second [11] that causes an axonal delay of a few milliseconds [12] for a spike to reach *efferent* neurons from the *afferent* one. In comparision, packets simulating spikes in the SpiNNaker chip move in nanoseconds. To achieve correct real-time results as that of biology, we need to apply this weighted input to accumulate into the stimulus to the neuron during its correct real-time interval. We need to store the delay between current time and the correct application time so that the input received now can then be applied at that time.

If we store the synaptic weight associated with each connection with the help of a 16-bit fixed-point number and axonal delay with 4-bits in a hash table containing 11 bits neuron's index in each processor as the key to the table, we require about 4-Bytes for one entry of the hash table to represent each connection [9]. For 1000 neurons in a processing core, each having 1000 connections [13], we require 1000x1000=1,000,000 words (4-MB) of data. The embedded processors are constrained for their local memories to only 32KB(instruction)+64KB(data) memory. For a chip with 20 such processors, we require a memory storage for minimum 4x20=80MB. To alleviate this problem, we store the synaptic information for each embedded processor's neurons in the SDRAM(up to 1GB). The DMA with each processing core provides a notion of localised memory to each embedded processing core while dealing with this data. The third event in this application is from the DMA that indicates to the processor that the relevant data is now loaded for a particular efferent neuron into the processor's local data memory and it can now proceed with its execution on this data.

Due to these events the SpiNNaker application is an event-driven real-time embedded application [14] that is quite different from typical sequential or multiprocessor parallel applications designed for parallel architectures. The following sections present the standard application model along with certain important architectural features and constraints that need to be kept in mind for an optimal application development.

1) Standard Application Model: Spike received event is generated by the Communication Controller on receipt of a multicast packet destined for neuron(s) in its processing core. The Communication Controller sends an interrupt to the Interrupt Controller to pass on to the processor. The Timer is responsible for the millisecond time event interrupt, while the DMA generates a completion interrupt after the synaptic data for relevant neuron(s) is made available into the processor's local data memory for neural processing. SpiNNaker uses the ARM Vector Interrupt Controller PL190 that can be configured with the help of the software for interrupt priorities. 16 out of 32 interrupts can be configured as vector interrupts i.e. the Interrupt Controller provides the address of an interrupt service routine to handle the relevant interrupt to the processor for fast execution of the interrupt service routine. The SpiNNaker address space has been distributed in such a way that the Vector Address Register in the Interrupt Controller containing the address of relevant interrupt service routine is not more than 4K away from the interrupt vector table for the IRQ interrupt. This enables the software to read the vector address register in the Vector Interrupt Controller in only one CPU cycle. We propose assigning the highest priority to the spike receive event, followed by the DMA completion interrupt and the Timer's millisecond interrupt. The event-driven real-time application model has been implemented with the help of interrupt service routines. On receipt of an IRQ, the start address of interrupt service routine is read by the embedded processor

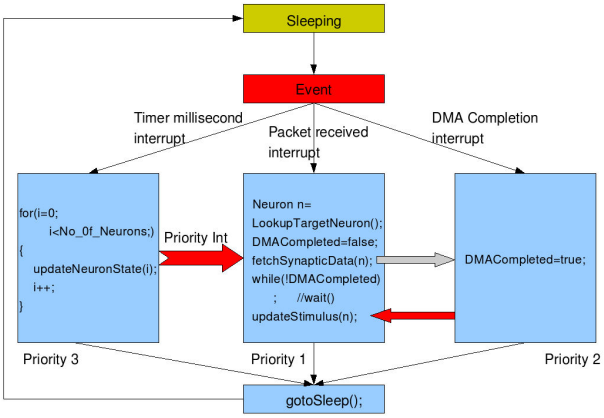

Fig. 4. Standard Application Model.

from the Vector Address Register of the Interrupt Controller. The functions responsible to update the projected stimulus and the state of neurons membrane potential are called from the relevent interrupt service routines as shown in Figure 4. The interrupt service routine for packet received interrupt requests a DMA read operation to bring in the synaptic data for the relevant neurons from the SDRAM to update stimulus for a projected time slot. On DMA completion IRQ, the interrupt service routine calls a function to update the stimulus for projected time slice at an axonal delay distance from the current time by adding the synaptic weight into the stimulus. The interrupt service routine on the timer interrupt will update the membrane potential for all the neurons simulated by each embedded processor. The default code called at the end of each interrupt service routine puts the processing core to sleep mode to conserve power.

2) Architectural Features Supporting the Application: Almost any spiking neural simulation model can be implemented on SpiNNaker, provided it is at a reasonable computational complexity level or the scale of the simulation may have to be compromised. The standard application model suggests that the architectural characteristics of the SpiNNaker chip should be kept in mind to fully utilize the power of its design. A few important features of SpiNNaker architectures that need to be made use of during application development are summarized here. SpiNNaker is capable of running highly parallel distributed applications on a million plus embedded processing cores generating tremendous system level throughput. The system has a very fast Spiking Communication Network organized as a toroidal mesh to support the spike transmission in a fraction of biological time. The network can be configured dynamically under the control of a application to support running any kind of neural network. The network supports a number of packet types that can be used for application execution and management. The system has a large amount of memory distributed over the system to hold large amount of application data. The processing cores contain local fast memory for running critical code. With the help of DMA and a fast network on chip, the processing core gets a localized view of the whole memory. The system is very scalable as a varying number of chips can be connected to form a system scaled to a desired simulation. The system makes use of low-power embedded processing cores to conserve energy, a system that can run for long time in a real-time temporal space to support large scale neural simulation.

3) Architectural Constraints Affecting the Application: Besides aforementioned features, there are a few important aspects that need to be kept in mind for a judicious use of the chip resources. There is a significant amount of memory available in the system, however like the brain, the memory is distributed over the entire system that needs to be used as per the criticality of the code and data. We have a small local, but fast, memory with each embedded processor that should be used for running critical code maintaining the state of the neurons and its state values; the large size SDRAM can hold synaptic information to be loaded into the local memory as and when required. The router has been designed to do source based routing of the packets. Total number of entries kept in the router are only 1K. However, if we follow the guidelines for configuring the routing table as explained in [15], i.e. mapping the neurons onto the processors based on their connectivity and using default routing to the maximum, these entries are sufficient for large scale simulations. Virtually any kind of neural network can be simulated over the SpiNNaker, however, the design supports a biologically inspired spiking neural network that can be mapped over the SpiNNaker hardware and simulated with the help of its standard application model in a comparatively easy way for better performance. As the application is running distributedly, it is hard to synchronize it at the system-level. The only way this can be done is by sending messages among the chips and to/from the Host System using small packets. For a real-time simulation to run over a hard real-time system, it is important to devise an efficient code to meet the critical time limits i.e. milliseconds during which all neurons inside a processing core have to update their state.

IV. Boot-up Process

Configuring the SpiNNaker CMP system involves booting-up each chip asynchronously while synchronizing the process at the complete system level. The novel architecture of this chip presents many challenges for an efficient configuration and management of the system. Some of these challenges are listed below:

a. The SpiNNaker chips do not have a dedicated hardwired monitor processor. For the chip-level management (and running an embedded Operating System), we require at least one processor out of the 20 on-board to perform such activities. the main role of the monitor processor is to help in preventing faults and if some occur due to unavoidable circumstances, these shoud be handled without disrupting the application. The monitor

processor is also responsible for running embedded code for chip-level configuration and testing before an application can run on the chip and later supporting the application running on other processors (fascicles). It is responsible for chip-level support functions like configuring the routing tables on the fly etc. It is also the responsibility of the monitor processor to maintain a P2P communication with the Host System outside the system and with other chips for system-level synchronization and management. For all these reasons, the code embedded into the Boot ROM needs to select, at run-time, a processor in every chip to perform as a monitor processor.

b. All the SpiNNaker chips are identical in every respect with no dedicated address hardwired. For the P2P communication to take place among the chips and that to the Host System, we need to assign addresses to these chips before the start of an application. For spike traffic to function properly and the router to direct these packets to their correct destination, we need to have chip address as part of the source ID of the firing neuron. The bootstrap code is responsible to assign a unique address to each chip after the system starts.

c. For want of better fault-tolerance the size of Boot ROM has been kept very small to avoid a single point of failure. For this purpose the bootstrap needs to be very small in size just to be able to perform chip-level Power On Self Test (POST), some functional tests run to check the application level functionality before the chip can run the application, and configuration of chip devices before the application can use them.

d. SpiNNaker needs to be attached to a Host System to interact with the system. The Host System is a normal PC for configuring the application before loading into the SpiNNaker system and interacting at run-time to provide stimulus to the system from the outside and to receive the responses before transforming these to some meaningful presentation to the user. This requires some way of connecting the system to the Host System. Though every chip is provided with an Ethernet Interface, only one (or a few) would be connected to the Host System. A protocol needs to be devised to connect the system to the Host System by enabling Ethernet Interface(s) only in relevant chips and making this communication possible in an efficient way.

e. A tailored spiking neural simulation is not embedded into the chip and can simulate varying behaviours of a particular neural network. The application is configured outside the system along with the neural mapping and connectivity configuration, which is then loaded into the system and dovetailed with the interrupt service routines to run as an application. This needs a detailed protocol to load the application along with connectivity configuration and synaptic information for each chip in an efficient and scalable way.

f. After the application is running, we need to communicate interactively with the system to see the state of the hardware devices and the application running on the chips. It needs a way to interactively communicate with each chip's monitor processor to provide inside information to the Host System. It also requires a common language to establish a meaningful communication among the chips and then with the Host System using small packets.

To handle these challenges, we have devised a very detailed protocol to help configuration and management of the system at chip- and system-level. Basic objective of the configuration process is to provide an efficient and fault-resilient way to bring each chip and thus the whole system to a functional state to enable running large scale neural applications in the most efficient way. Yet another objective of this research is to provide a way to configure the system in the most scalable way i.e. the configuration process may be affected by the size of the system and amount of loadable data size to the minimum. Important steps of the protocol have been enumerated below:

- Perform a POST on every processing node and its peripherals.
- Select a monitor processor out of the healthy processing cores in each CMP.
- Perform detailed chip-level device tests followed by a few functional tests to check the correct functionality of chip resources.
- Perform chip-level recovery if part of the chip is non-functional.
- Test connections to the neighbouring chips and mark any faulty link or neighbour to avoid congestion by avoiding packets sending to that direction.
- Attempt to cure an unhealthy chip with the help of its neighbours (neighborhood probing).
- Load a micro-kernel in each monitor processors memory from the Host System using broadcast NN packets.
- Assign addresses dynamically to the chips to enable inter-chip point-to-point and multicast communication.
- Report the health of each chip resources to the Host System to enable it configuring the application.
- Load the application to each chip using a broadcast mechanism.
- Configure the routing tables for each chip to enable routing.
- Replace an unhealthy monitor processor with another processing core at the runtime and reset the chips or their resources if not working properly.

V. Application Loading Process

A neural network simulation application along with neuron's mapping i.e. placing neurons on individual embedded processors and the connectivity information needs to be loaded into the system with the help of embedded boot-up code. Along with the application, we need to load the synaptic data into the SDRAM of each chip to be used by the application during its execution. To establish neural

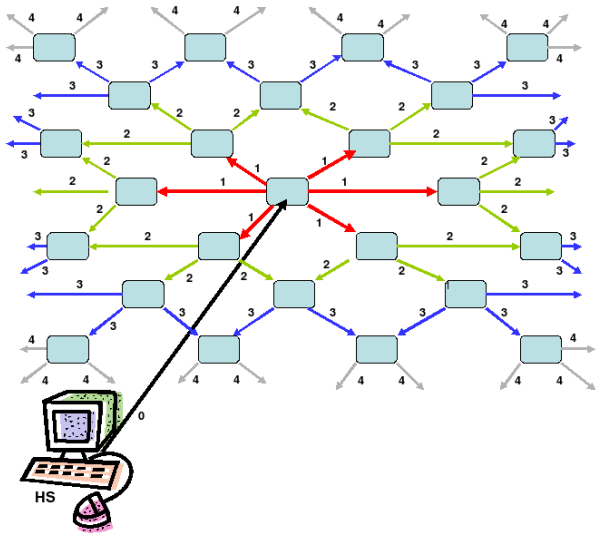

Fig. 5. Selective Forward Flood-fill.

connectivity, we need to configure the routing tables at each chip to formulate a neural network. Besides the neural application to run on fascicle processors, we need to load a micro-kernel and a utility functions library to each chip's System RAM to be used by the monitor processor. This requires a large amount of data to be loaded on the system from the Host System with the help of Ethernet connection. We plan to establish Ethernet connection between the system and the Host System from only one or at most 4 chips. The connection with more than one chip is for redundancy to achieve better fault-tolerance. From the connected chip to the rest of the system, the data is loaded with the help of Communication Network that connects each chip with its six neighbours. The communication between the Host System and the connected chip is Ethernet Frame based (upto 1500-B data) while the communication network uses only small packets (4-B data). We need an efficient way to load this data or it may take a very long time before the application could be started on the system. An efficient and fault-tolerant mechanism has been devised to load the application into the system. The NN type of packet is used to broadcast the data to the 6 neighbouring chips sending one word (32-bits) at a time. The receiving chips store the data if not received so far and broadcast it to its neighbours. This way a pipelined wave of broadcast data flows from the origin(s) to the whole system and wraps it around. As part of this flood fill mechanism protocol, an instruction set has been created to pass meaningful messages between the monitor processors on the neighboring chips. Address space 0x0F800000 - 0x0FFFFFFF in the SpiNNaker address space has been dedicated to contain these instructions in the NN packet address field. Once an NN packet with an address within this range is received, the monitor processor on the other sides takes it as a special instruction as part of NN conversation protocol. Figure 5 shows one of the selective forward multicast mechanism for flood-filling the data in the SpiNNaker system. The process of application loading into the system works as follows:

- The application and the data is loaded to the connected chip(s) one frame at a time.
- The frame contains the size of the data block along with block level checksum for error detection. The connected chip performs a checksum test on the block to accept or reject it. If no errors are detected, the data block is stored in the processor's local data memory.
- The origin chip sends an instruction as part of NN packet to indicated the size of block it is about to transmit and the starting address of the block destination in the SpiNNaker chip address space i.e. processor's tightly coupled memory, System RAM, Router or SDRAM.
- Each neighbouring chip dedicates a memory space in its data tightly coupled memory equivalent to the size of data block being received. It also initializes a bitmap with each bit respresenting a word in the block to indicate receipt of a word with bit value '1' in order to control duplication. If a word has already been received, it will neither be stored in the memory buffer nor will be transmitted further.
- The physical address of the word is sent along with the NN Packet in the routing key, again to control the duplicate words.
- The last NN packet to transmit a block of data contains an instruction indicating that a checksum is being passed as the end of block level transfer.
- If the block passes the error detection test, it will be loaded into the specified location in the memory address space.
- If the passed-on block contains the application, it will be loaded into the instruction tightly coupled memory at a particular stage of the process and the control of execution will be handed over to this application.
- Each chip receives a transmitted word at least twice during the flood fill process for redundancy, in case a link is broken or congestion on one side prohibits the packet transmission from one neighbour.
- At the end of flood fill process, the host system asks for the state of each chip along with blocks received. At this stage, the chips can request from each other or the Host System for any missing block.
- During neighbouring chip recovery process, the data is locally transported by the neighbouring chip after a faulty chip is repaired.

VI. Hardware Abstraction Layer

Despite the architectural intricacies, we provide a simple user interface to support software development for the SpiNNaker system. A Hardware Abstraction Layer (HAL) has been implemented as part of this research with the functions

to support these events and to update neurons states as a result of these events. At low level, device drivers for all the chip components like Communication Controller, Timer, Vector Interrupt Controller, PL340, Multicast Router, DMA Controller, System Controller and Watchdog Timer have been provided. Besides these some middle layer functions to support inter neuron's communication have been provided to send/receive spikes. The abstraction layer provides a localised view of the whole memory to the user by handling DMA operations in an efficient way. To support the event-driven application model, interrupt service routines associated with these events have been implemented with entry points for the high level application functions to perform neural state and stimulus updates. Some high level functions to support a sample application as explained in [9] have been implemented as a guide to the developer. All these functions have been implemented in ARM Assembly for ARM968E-S for optimal performance. It is because the embedded real-time application has hard time bounds of milliseconds to update neural state to support a real-time simulation of biologically inspired spiking neural simulation model. The functions have been compiled in a library to be available to the developer for application development without exposure to the architectural intricacies. Library functions' signatures have been provided to the application developers for easy portability of applications. The developers can use their own definition of these functions with the same signatures while developing the application on a PC or a cluster computer with the similar functionality. Later the same application can be compiled by including the SpiNNaker HAL library to port it to the SpiNNaker architecture. It is to support early development of the software while the hardware is still in design phase.

A user friendly interface at the Host System to configure, load and interact with the application running on the system, as if it is running on a normal PC, is being developed as part of this research. The interface shows the state of the system as a graphical view and helps the user to interact with the system. A sample application will be provided with the interface as a tutorial for running the spiking neurons application on SpiNNaker.

VII. Verification

The configuration process has been implemented using ARM Assembly code and tested on a SystemC based system-level model of a single- and multi-chip SpiNNaker system developed using ARM SoC Designer. The model uses a cycle accurate instruction set simulator for ARM968 processor from SoC Designer along with the models for other ARM provided components such as Interrupt Controller PL190, PL340 SDRAM Controller, Watchdog Timer Controller, AHB bus, APB Bus and memories. All the components provide cycle accurate behaviour and have been tested by ARM. SystemC models for indigenously designed components were prepared as part of this research [16] like those for the MCRouter, Communication Controller, Communication NoC, asynchronous Tx and Rx interfaces and System Controller etc. The model as a whole provides a cycle approximate behaviour as expected from a SystemC Transaction Level Model (TLM) of a system-level model. The model simulates 3 processing cores per chip as expected in the test chip for simplicity of simulation on the host PC with limited memory. The code can be tested for performance and debugged for semantic correctness. All the memory locations can be viewed to verify the code execution correctness. The configuration code was loaded into the Boot ROM and after the enabling of the processors' tightly coupled memory was loaded into ITCM.

TABLE I
Boot-up Process Time

No. of Chips	No. of Procs.	No. of Cycles	Cycles/sec
1	1	305712	40415
1	2	322507	32609
1	3	334696	31839
3	3	334712	3633

The code execution time is not dependent on the number of chips in the system as it runs concurrently on all the chips. However, the number of processors on the chips do affect overall time as the code needs to be copied to the processor's local memory from the Boot ROM and because all the processors will be copying from the same source, it will delay the process due to the contention while accessing the system NoC as shown in Table I. Column 3 shows the CPU cycles of ARM968 processor used to execute chip level boot-up code, while column 4 shows the speed of simulation on the host PC. For SpiNNaker chip, we are able to run ARM968 processors at 200MHz.

Loading the application using Ethernet Frames from the Host System to the connected chip was also tested on this system-level model with one chip simulation. However, as the model can not simulate a large scale system due to its computational complexity, the flood-fill process was tested on a higher level abstract model simulating only the Communication Network with the help of timing related to send and receive of packets obtained from the detailed system-level simulation. The results of the simulation are shown in Figure 6. The results show that the flood-fill process is not dependent on the size of the system. This is because each chip passes on the packet to its six neighbours and for a large size of application like 100KB, the number of hops to reach the farthest point from the origin in the (256x256=64K chips) system is neglegible compared to 25K packets. The results also show that the process does not take long for a large size of data to be filled in. One surprising result is that the flood-fill process is not dependant on the number of Ethernet links to the system as there is hardly any performance gain with 4 Ethernet links over the one. There is however a considerable gain of performance if we adapt selective multicast to send packets to the forward direction instead of broadcasting to all the six neighbors that causes congestion. It is very logical as we should not be sending the packets back to the neighbours we are getting these from. However, we need to be very

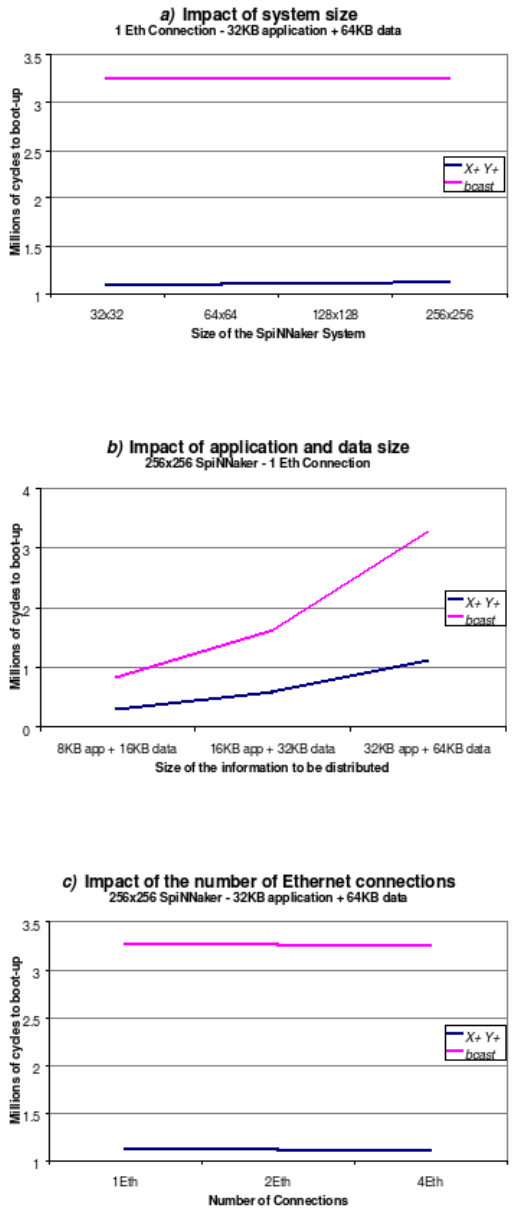

Fig. 6. Application Loading on SpiNNaker a, b, c.

selective to provide reasonable redundancy of the packets being sent so that a chip may not use a data block even if only one word is missing due to some transmission problem. We need to maintain a logical balance between the performance and redundancy (fault-tolerance). We can compromise on performance in favour of fault-tolerance at this stage as the application and the data is very critical to the overall simulation.

An event driven model for target application using Izhikevich model as explained in [9] has been implemented using this configuration process. The application runs fine updating the neural state in millisecond temporal dimension while sending and receiving spikes as per the neural dynamics model. With these results we are quite confident in the design and functionality of the SpiNNaker system and expect the system will be very useful to the research community to exploring the mysteries of human brain.

VIII. Conclusions

We are researching a novel technique to configure a large-scale CMP system with a high degree of efficiency and fault-resilience. To avoid a single point of failure, the selection of monitor processor in a chip and chip address assignment has been kept soft. One processor is chosen out of 20 on-board embedded ARM968s to perform as a monitor processor for managing the chip resources and synchronizing the application at the system-level. A mechanism has been designed to assign virtual addresses to all chips for inter-chip communication. All chip components will be configured, tested and accessed with the help of embedded device drivers' code written specifically as part of this research. We provide a library of functions to support application developers to develop spiking neuron simulation applications without exposure to the architectural details of SpiNNaker SoC. A small size Boot ROM contains code just enough to test and configure the chip resources. Neighbouring chips can examin each other's resources for diagnostics and recovery purpose using NN packets. Thus a chip can still boot-up without a Boot ROM with the help of its neighboring chips. A novel flood-fill mechanism has been devised to load large-size applications on the chips from a Host System outside the SpiNNaker system. The mechanism loads the application quite efficiently, independent of the size of the system using a NN packets broadcast mechanism. A cycle- and instruction-accurate SystemC based system-level model for the SpiNNaker SoC has been prepared to verify the design objectives. The simulation shows that the chip can be configured very efficiently and made available to the application. A high-level simulation for the inter-chip communication network suggests that application loading into the system is very efficient, independent of the scale of the system.

A user-friendly software interface has been designed to interact with the system from the Host System. The user interface running on the Host System would help in configuring the system, debugging a particular hardware component, and running a neural simulation in an interactive way. Presently we are working on refining the flood-fill mechanism to load the application with a selective forward multicast instead of broadcasting the NN packet to all the neighbours for better performance as suggested by the simulation. We are also working on refining the user interface to help configure the application before loading into the system. We are continuously interacting with the application developers at the University of Southampton to find the best way to map neural networks on the system and to optimally configure the routing tables for inter-neuron connectivity.

ACKNOWLEDGEMENTS

The SpiNNaker project is supported by the Engineering and Physical Sciences Research Council, partly through the Advanced Processor Technologies Portfolio Partnership at the University of Manchester, and 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. Mr. Javier Navaridas is supported by a doctoral grant of the UPV/EHU and by the Ministry of Education and Science (Spain), grant TIN2007-68023-C02-02.

REFERENCES

[1] S. Furber, S. Temple, and A. Brown, "On-chip and inter-chip networks for modelling large-scale neural systems," in *Proc. International Symposium on Circuits and Systems, ISCAS-2006*, Kos, Greece, May 2006.

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

[3] T. Elliott and N. Shadbolt, "Developmental robotics: Manifesto and application," *Philosophical Trans. Royal Soc.*, vol. A, no. 361, 2003.

[4] *SpiNNaker - a chip multiprocessor for neural network simulation*, 0th ed., The University of Manchester, Nov. 2007.

[5] 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.

[6] L. A. Plana, S. B. 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*, vol. 24, no. 5, pp. 454–463, Sept.-Oct. 2007.

[7] L. Plana, J. Bainbridge, S. Furber, S. Salisbury, Y. Shi, and J. Wu, "An on-chip and inter-chip communications network for the spinnaker massively-parallel neural net simulator," in *Proc. Second ACM/IEEE International Symposium on Networks-on-Chip (NoCS 2008)*, 2008, pp. 215 – 216.

[8] S. B. Furber, S. Temple, and A. D. Brown, "High-performance computing for systems of spiking neurons," in *AISB'06 workshop on GC5: Architecture of Brain and Mind*, vol. 2, Bristol, April 2006, pp. 29–36.

[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] E. Izhikevich, "Simple model of spiking neurons," *IEEE Trans. on Neural Networks*, vol. 14, pp. 1569–1572, Nov. 2003.

[11] R. F. Thomspon, *The Brain - A Neuroscience Primer*, 2nd ed., G. L. R. C. Atkinson and R. F. Thomspon, Eds. New York: W.H.Freeman and Company, Worth Publisher, 1997.

[12] J. A. G. E. M. Izhikevich and G. Edelman, "Spike-timing dynamics of neuronal groups," *Cerebral Cortex*, vol. 14, no. 8, pp. 933–944, 2004.

[13] H. Markram and M. Tsodyks, "Redistribution of Synaptic Efficacy Between Neocortical Pyramidal Neurons," *Nature*, no. 382, pp. 807–810, 1996.

[14] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[15] 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.

[16] M. Khan, X. Jin, S. Furber, and L. Plana, "System-level model for a GALS massively parallel multiprocessor," in *Proc. 19th UK Asynchronous Forum*, London, Sep. 2007, pp. 9 – 12.