ARTICLE IN PRESS

Parallel Computing xxx (2011) xxx–xxx

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric

M.M. Khan [a], A.D. Rast [a,*], J. Navaridas [a], X. Jin [a], L.A. Plana [a], M. Luján [a], S. Temple [a], C. Patterson [a], D. Richards [a], J.V. Woods [a], J. Miguel-Alonso [b], S.B. Furber [a]

[a] School of Computer Science, The University of Manchester, Oxford Road, Manchester, M13 9PL, UK
[b] University of The Basque Country, Manuel Lardizabal, 1 20019 San sebastián, Spain

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Chip multiprocessor
Configuration
Asynchronous
Boot
Neural
Network

ABSTRACT

Configuring a million-core parallel system at boot time is a difficult process when the system has neither specialised hardware support for the configuration process nor a pre-configured default state that puts it in operating condition. The architecture of SpiNNaker, a parallel chip multiprocessor (CMP) system for neural network simulation, is in this class. To function as a universal neural chip, SpiNNaker uses an event-driven model with complete system virtualisation so that all components are generic and identical. Where most large CMP systems feature a sideband network to complete the boot process, SpiNNaker has a single homogeneous network interconnect for both application inter-processor communications and system control functions. This network improves fault tolerance and makes it easier to support dynamic run-time reconfiguration, however, it requires a boot process compatible with the application's communications model. Here, we present such a boot loader, capable of bringing a generic, initially unconfigured parallel system into a working configuration. Since SpiNNaker uses event-driven asynchronous communications throughout, the loader operates with purely local control: there is no global synchronisation, state information, or transition sequence. A novel two-stage "unfolding" boot-up process efficiently configures the SpiNNaker hardware and loads the application using a high-speed flood-fill technique with support for run-time reconfiguration. SystemC simulation of a multi-CMP SpiNNaker system indicates an error-free CMP configuration time of ~1.37 ms, while a high-level simulation of a full-scale system (64 K CMPs) indicates a mean application-loading time of ~20 ms (for a 100 KB application), which is virtually independent of the size of the system. Further hardware-level Verilog simulation verified the cycle-accurate functionality of CMP configuration. The complete process illustrates a useful method for configuring large-scale event-driven parallel systems without having to provide dedicated hardware boot support or rely on system state assumptions.

© 2011 Elsevier B.V. All rights reserved.

1. Introduction

Flexible and efficient boot loading of distributed applications is an essential support process for any large-scale parallel computing system. In the case of the SpiNNaker system, a massively parallel platform connected over an homogeneous, asynchronous communication fabric, the absence of any dedicated sideband network or system synchrony presents additional challenges. The system must somehow break symmetry, assign and load memory resources, configure communications,

* Corresponding author.
E-mail addresses: khanm@cs.man.ac.uk (M.M. Khan), rasta@cs.man.ac.uk (A.D. Rast).

0167-8191/$ - see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2011.02.003

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

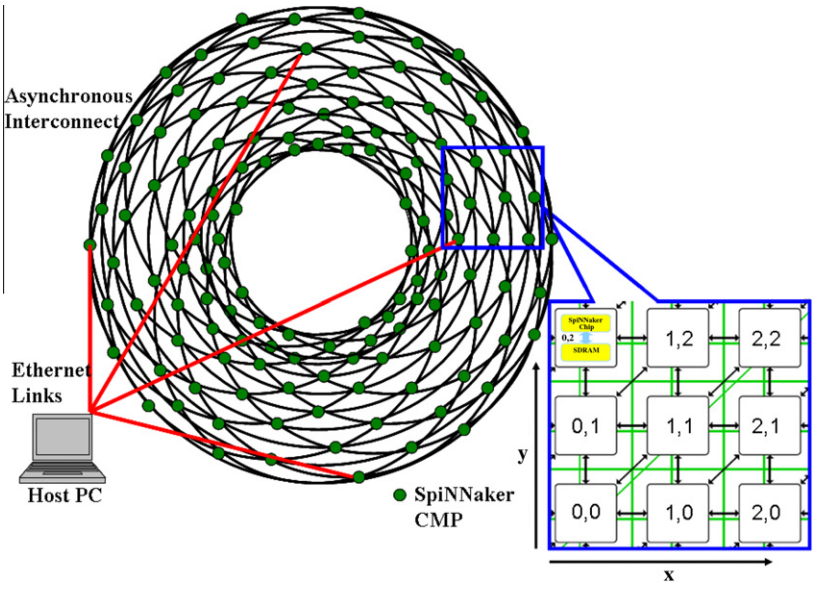


Fig. 1. Multi-CMP SpiNNaker system forming a 2D Toroidal Network.

and start up the processors, while balancing concurrency and resource contention for maximum efficiency. Where previous solutions [8,14] typically use sideband communications or dedicated preconfigured resources, SpiNNaker confronts the challenge of configuring an isotropic undifferentiated parallel processing system head-on.

One approach would be to make no assumptions about the application and consider it as a problem in general-purpose computing, leading to a set of standardised, generic configuration techniques. However, since numerous studies indicate that parallel processing works best with specific applications having inherent parallelism [5,13], it seems reasonable to design parallel systems around a target application, whose boot process could be correspondingly specialised. SpiNNaker is a chip multiprocessor (CMP) for one such application: large-scale spiking neural networks, whose architecture is inherently parallel. This is a "match-fit" candidate application for distributed processing systems: indeed, the consensus in the modelling community is that it may be necessary to use dedicated hardware with architectures more closely similar to the biology for large-scale neural modelling within realistic resource limitations [15]. It is efficient to simulate a spiking neural network as an event-driven real-time application [16], a model quite different from typical parallel applications and more akin to embedded applications [19]. A system for neural network simulation will be, correspondingly, architecturally different from parallel systems designed mostly for general-purpose computing. Current-generation neural systems (such as SpiNNaker) adopt mostly event-driven models of computation, with boot-time configuration considerations that can make fewer assumptions about the initial state of the system than "conventional" parallel multiprocessor systems.

SpiNNaker provides no sideband communication channel for boot processes: the system boot must use the same communications fabric as the application. All processors on the chip are identical; there is no dedicated processor hard-wired or preconfigured to run the boot process. The network interconnect is an homogeneous asynchronous network, with no predefined origin or start point, connecting identical CMP's (Fig. 1). The task, therefore, is as follows: configure a symmetric, massively parallel system using only the resources available at run time, even though the functionality of these resources themselves depends upon having been configured. The configuration process must do this efficiently and without generating resource conflicts, even though individual processors have only local state information available, i.e. the system can use no global state information to configure itself.

We demonstrate here an efficient method for configuring the SpiNNaker system, based on the resources available at run time. The essence of the method is a two-phase process operating as follows. First, a random process selects one processor per chip as a "monitor" processor (MProc) and likewise a reference chip in the multi-CMP SpiNNaker system. This breaks the symmetry of the processors on-chip and the interconnect between chips. Then the MProc completes the configuration, broadcasting packets over the network to distribute the neural network configuration from the newly defined central source point to all the chips. The method demonstrates a useful way to leverage inherent asynchronicities in both device and application to achieve a fast boot process in an application-specific parallel system.

2. Existing large-scale configuration methods

Here we present the configuration process in large-scale massively-parallel systems to examine how the configuration issues are handled in these architectures to support large-scale parallel distributed applications. Relatively little previous work

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

has been done to address the need for boot loading within an initially unconfigured parallel system, in part because previous systems often use standard, fixed hardware for which the conventional technique of a dedicated BIOS and boot-loader work adequately. Most other systems, in fact, rely on the availability of some form of dedicated resources or pre-existing low-level boot support in order to get up and running. Two architectures similar to SpiNNaker are large-scale multi-CMP systems interconnected in a torus by connecting each chip to six neighbouring chips using a packet-based network.

IBM's Blue Gene/L (BG/L) supercomputer's computational core comprises 64 K nodes, each consisting of 2 PowerPc400s running at 700 MHz. One of these is used for user applications while the other is used for configuration, networking support and chip management. Each compute node has six bi-directional links connecting it with other compute nodes, interconnected using a $64 \times 32 \times 32$ three dimensional torus [1]. A *sideband infrastructure* is used for configuring, controlling and monitoring the system. This configuration subsystem consists of service nodes, communicating with the compute nodes using packets over a separate dedicated Ethernet. There is no Boot ROM in the compute nodes; rather, the whole initialization is done at run-time by the service nodes through the sideband Ethernet [14]. A small boot loader is first written directly to each compute node's local memory which receives and loads the boot image into the memory using packet-based messages from the service nodes [4]. After this common information, each node is "personalized" by assigning distinct torus coordinates, a tree address and an IP address. The boot process in Blue Gene/L is based on the concept that the service node can control the computational core to the lowest level of granularity [14]. Service nodes use the Ethernet sideband network to drop packets to a control FPGA chip attached to each node card in order to configure the compute nodes. The control FPGA converts the packet based communications to JTAG (IEEE 1149.1) commands for the compute nodes. These packets contain the common boot loader and boot image. Similarly, information such as faults or acknowledgements, etc., is transferred back to the service node by these control FPGA chips in the form of packets. During initial built-in self-test, chips failing to pass all tests are marked as dead chips.

The XT3 is Cray's third-generation massively-parallel processing system [2] designed to run large-scale parallel applications. This is a multi-node system connected in a 3D torus through Cray XT3 interconnect and the Cray SeaStar routing mechanism. Its Compute Processing Element (PE) is an SoC based on a dual-core AMD Opteron processor coupled with its memory and dedicated communication resources. Another type of PE, the Service PE, is used for I/O, login, and network/system management functions [8]. Each Compute PE is attached to a SeaStar communication chip containing a DMA engine, hyper transport links, a communication and management processor, an interconnect router and service port. An on-chip router on the communication chip connects each node (compute node and communication chip) to six neighbouring nodes in a 3D torus topology. Each communication chip provides a service port which connects each Compute PE to a separate dedicated management network (a sideband Ethernet), and bridges between the management network and the system interconnect [8]. This allows the monitoring system to have access to all registers and memory in the system. The management network is used for boot-up, maintenance and monitoring of the system during both initialization and execution. The fast interconnect with no memory module contention provides a fast boot-up mechanism with minimum downtime [8]. The Cray configuration/management process integrates hardware and software components to support run-time application execution, system monitoring and fault-handling. An independent monitor system with separate processing and network resources monitors and manages major resources in the Cray XT3 system. The same system is used to manage the interconnect, to monitor and display the system state to the system administrator, and to power up and down the system. System-level recovery in the case of any malfunction is also controlled from the system management system. The management system does not use any system resources and recovers any failed component while the system remains running. Single-points-of-failure have been minimized with redundancy or alternative workarounds in the design, and hence the system remains operational even if a (non-critical) portion of it is non functional [3,7,8].

A popular approach in modern high-performance computing is cluster computation: exploiting the resources of a pool of standard PC's in order to achieve effective performance similar to a dedicated HPC system. Diesburg [10] conceives of the concept of the "Bootable cluster CD" – a distributable boot loader that auto-generates a cluster configuration from a series of initially unrelated machines. The system uses an overlay approach to run entirely in RAM. Central to the methodology is a series of sources resident on remote Internet (WWW-based) servers; this assumes the presence of a TCP/IP network. The system thus preserves the notion of initially undifferentiated, non-dedicated hardware but makes fairly strong assumptions about the network configuration and operating system support services already in place on the hardware.

3. SpiNNaker real-time NN simulation model

3.1. Biological information processing model

Because SpiNNaker is a domain-specific design, many of its features emerge from the requirements of its target application: simulation of biologically realistic neural networks. It is useful to discuss this briefly to place in context the features and functionality available to the boot process which is the subject of this work. Characteristically, neural networks are massively parallel and highly interconnected, a virtual "match fit" for a parallel distributed processing system [15,21,23]. Individual processing elements (neurons) have no global system view; they must use only local information to compute updates. This in turn means that there is no need for memory coherence within the system. Biological neurons communicate through spikes: short-duration impulses [9]. It is usual to abstract the spike to an instantaneous event triggered when the neuron

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

reaches a certain threshold value [20]. Biological time scales are relatively slow: a time resolution of 1 ms per neuron is adequate [24], and typical spike rates are on the order of 10 Hz with an upper bound of about 100 Hz. Neurons typically have high fan-in: ~1000–100 K inputs per neuron [6], but sparse activity: ~0.01–1% active neurons in a given population. Any given neuron might then expect 100 events/s in normal operation; maximum event rates are usually no higher than 10,000 events/s. Such event statistics make it practicable to use a "wake-on-event" processing model within SpiNNaker, where individual processors remain asleep until activated by the arrival of an event – a spike.

3.2. Hardware support

The SpiNNaker CMP (Fig. 2) is a System-on-Chip (SoC) architecture with multiple (18) processing nodes connected through an asynchronous Network-on-Chip (NoC). While the processing cores are general-purpose CPU's, SpiNNaker's design optimises the functionality for running spiking neural network models in real time [11]. Each processing node comprises an ARM968E-S processing core and supporting peripherals: an interrupt controller (VIC), timer, DMA controller and communications controller. With its fully programmable design, SpiNNaker can implement almost any arbitrary spiking neural simulation model within a tradeoff envelope defined by model computational complexity, number of neurons simulated, and real-time update performance requirements.

Chips connect together into a large-scale system whose physical topology is a hexagonally-connected toroidal mesh (Fig. 1). The inter-CMP fabric (the Communications NoC) itself uses a packet-based protocol with four different (40–72 bit) packet types: neural multicast (MC), point-to-point (P2P), fixed-route (FR) and nearest-neighbour (NN). MC packets are the primary neural spiking type and are source routed using programmable entries in the on-chip router. The P2P packet is a chip-to-chip (as opposed to processor-to-processor) type used for system-level management, while the FR packet is mostly for debugging and is automatically routed to the nearest Ethernet-connected chip. NN packets are central to the boot process and are used for chip-level diagnostics and configuration. A chip can send an NN packet only to adjacent chips: either to a specific neighbour, or to all neighbours. This type also allows the chip to "peek" and "poke" neighbouring chips' resources. The NoC uses a configurable on-chip router to support neural networks with arbitrary connectivity [18]. There is one further communication resource for off-system communications: an on-chip Ethernet interface. While each chip has such an interface, in a real system only a subset of CMPs actively communicate to the external user interface device, usually a PC workstation (called the Host PC).

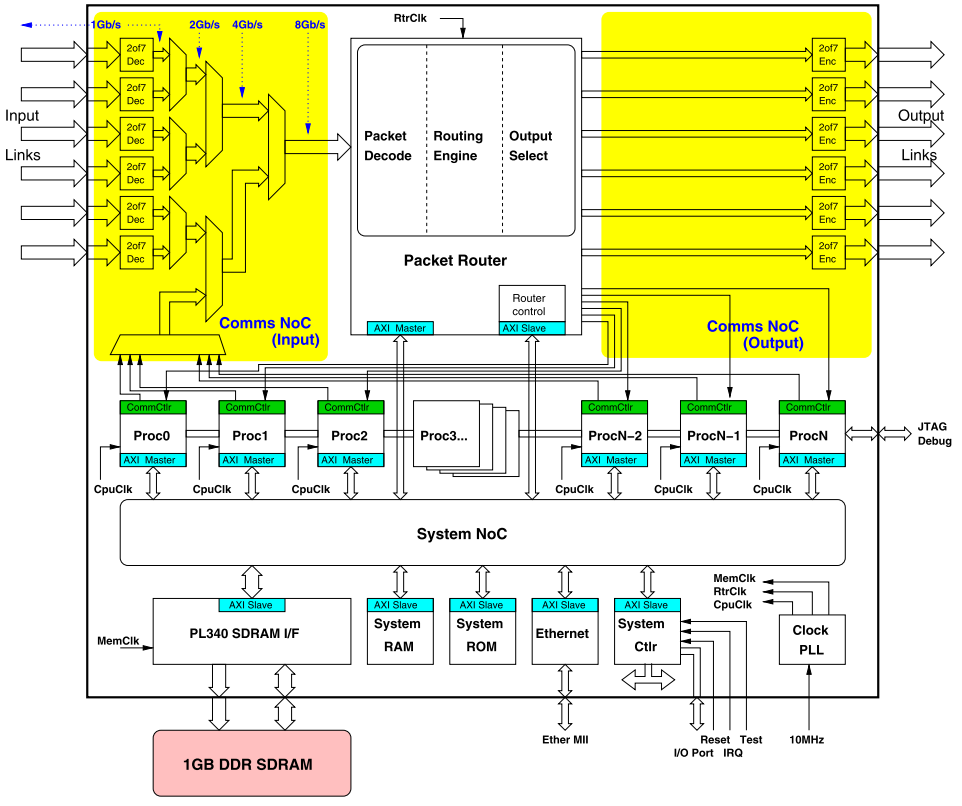


Fig. 2. SpiNNaker CMP.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

SpiNNaker has a hierarchical memory system: each processor has only a small amount of local memory in a 32 KB instruction and a 64 KB data tightly-coupled memory (TCM). Supplementing this is a high-speed on-chip SRAM, shared between on-chip processors. Main memory lies off-chip in a 128 MB SDRAM (one per chip), accessed through a proprietary DMA controller over a separate NoC (System NoC).

Each SpiNNaker CMP contains an Ethernet interface to connect it with the outside world. Using the on-chip Ethernet, a given SpiNNaker CMP can be configured to present the entire internal network to the Host PC as a single, fully functional SpiNNaker system, simulating a neural network of up to 10^9 neurons (using a reference neural model). Alternatively, a multi-CMP SpiNNaker system can be divided into more than one subsystem to run different neural simulations concurrently, by configuring its routing tables to partition the system and connecting each partition to the Host PC using different Ethernet connections.

3.3. The SpiNNaker execution model

SpiNNaker's processing and communications are event-driven. Within the system, a set of events makes it possible to provide fast interrupt-driven hardware support for neural processing while minimising the detailed low-level knowledge the user needs to run a given neural model. An on-board configurable vectored interrupt controller provides hardware support to prioritise interrupts. Three interrupt sources within each processor: the timer, communications controller, and DMA controller, are central to the model (Fig. 3). The timer triggers a "millisecond" interrupt to schedule neuron update, while the communications controller issues a "packet-received" interrupt upon receipt of a spike. The DMA generates a "DMA-complete" interrupt upon completion of a background synaptic data transfer to/from SDRAM, initiated on receipt of a spike.

Several features of the design are important in considering the boot process. Since most on- and off-chip processes are event-driven, there is no global inter-process synchronisation and the boot process cannot rely on fixed timing relationships between processors. In addition, all dynamically updated state information must fit into the small 64 K data TCM local memory. While each SpiNNaker chip has a large memory resource in the SDRAM, it is not accessible until it has been configured. During application execution, DMA operations effectively "hide" the non-local nature of the SDRAM [22], but at boot time, this is not the case – and in fact the SDRAM represents another uninitialized, unmapped memory resource. Similarly, the router is blank at start-up and thus the boot process can rely only on the default routing mechanisms to configure the system, while at the same time needing to load the routing tables. The communications network can use only NN packets at boot time, since P2P, FR and MC packets require configured routing tables. The boot configuration process must use the Ethernet interface as an entry point to the system network, but it does not provide any visibility to the SpiNNaker system beyond the chip(s) directly connected to the Host PC. The boot loader must therefore be quite different than in most large-scale systems, which power on with a greater amount preconfigured or with more known, and which can usually provide better global system visibility.

4. System configuration and application loading

4.1. Configuration challenges

The boot process needs to configure each chip in a SpiNNaker system individually, after which it operates as an integrated computing system. SpiNNaker is characterised by an homogeneous architecture at the chip and system level. At the chip

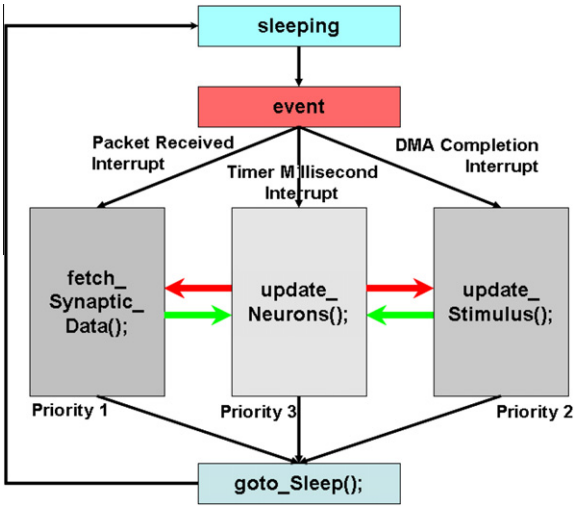


Fig. 3. SpiNNaker event-driven neural application model.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

level, it has an array of identical processing nodes, while at the system level, chips are connected in a closed asynchronous mesh network having no addresses or distinct reference nodes. This system symmetry creates the following challenges for the two phases of the boot process.

(a) *No Sideband Configuration Resources*: Most other multi-CMP system (such as Blue Gene/L or Cray XT3), as mentioned in Section 2, use a sideband network connecting all CMPs to an outside resource (such as a service node). This allows direct access to every chip's resources and helps in loading the boot loader, initializing chip resources, assigning chip addresses ("personalizing" chips), loading the application and initializing the inter-chip network. However, SpiNNaker does not use any sideband network, but rather the same homogeneous asynchronous network the application uses. For external communications, the Ethernet network is used to connect the Host PC with the SpiNNaker system (potentially through multiple chips for redundancy). This is a significant challenge for the configuration process because the state of other, unconnected chips is unknown and the network has not yet been initialised.
(b) *No Global Shared Information*: Each chip has only *local* information about its resources, clock speed, network connections and router state. There is no information about the size of the system, state of other chips, or the location of chip(s) connected to the Host PC. There is no central clock and chip activities are not synchronized. The system is somewhat analogous to the Internet, comprising many functionally independent, asynchronously operating computing devices. On startup, each chip has to start initializing independently with no global system information available. The challenge is then to configure the hardware to run distributed applications as one integrated system.
(c) *No Preconfigured Routing*: As each chip's router needs a different configuration, the boot process needs to access each chip individually and configure its routing tables. However, accessing an individual chip from outside of the system is impossible when the chips have no address and the network has not been configured, so a default routing method is needed.
(d) *No Fixed Connection Points to Outside World*: Any chip can be connected to the Host PC using the on-chip Ethernet connection. However, since we connect the system using only a subset of chips, the connection point information is unknown to the chips at startup. We need to devise a process to let a Host-connected chip know that it is a connection point and then to inform other chips about the route to communicate to the outside world.

4.2. Configuration requirements

Particularly at boot time, but also at run-time, SpiNNaker appears as a generic processor resource: a neural "blank slate". To enable it to load and run a given neural network successfully, there are, as a result, several key system configuration considerations.

(a) *Selection of Monitor Processor*: For chip-level management, we assign one processor out of the 18 on-chip to be the Monitor Processor (MProc). The MProc has three roles: pre-boot chip-level configuration and testing, chip-level fault handling, and system-level management support. It must perform these management tasks without disrupting the application processors running the neural application. SpiNNaker chips, however, do not have a dedicated MProc, therefore, the boot process needs to select a processor per chip to perform this job.
(b) *Breaking System-wide Symmetry*: To route a packet (P2P or MC) to its destination, the chips need unique addresses. SpiNNaker chips are identical, with no hardwired addresses, while the SpiNNaker system is organized as a symmetric toroid with no starting point (Fig. 1). Before any application can run on the SpiNNaker system, we need to configure the chips with unique addresses.
(c) *Run-time Configuration*: To conserve the CMP area, the Boot ROM size has been kept to a minimum, just sufficient to support initial testing, boot, and device initialization. The remaining CMP- and system-level configuration must be performed from outside the system, improving flexibility and fault-tolerance.
(d) *Run-time Application Loading*: SpiNNaker CMPs are not preconfigured to simulate a particular neural dynamic model; they can simulate arbitrary spiking neural models. This means the user initially configures the application outside the system, then loads it. As a result, we need a detailed methodology to load the neural application with associated data to each chip in an efficient and scalable way.
(e) *Network Configuration*: On-chip routers must be configured to route interneuron spikes at run-time according to the intended neural network application. The configuration process needs to configure all the on-chip routers before a neural application can run. However, a neural network is simply a configuration, not in itself a running application. Much like an FPGA, the boot process can configure the network once after which the target application can be run many times without reloading the system. Therefore although fast boot is desirable, it is not necessary to consider boot time as a function of the time to run a given application – the neural network is not a terminating "program".
(f) *Communication to Outside World*: SpiNNaker needs to be attached to a Host PC to load the neural application and interact with the user. Typically, the Host PC would be an industry-standard Windows/Linux PC, requiring some interface to the SpiNNaker machine. Every chip has an Ethernet MAC (Media Access Control) interface, but given that only a subset of chips have a PHY (Physical Layer Interface) and Ethernet connector to link it to the Host PC, there must be a protocol bridging Ethernet communication (between the Host PC and Host-connected chips) and packet-based communication (among the CMPs).

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

(g) *User-system Interaction*: Once the application is running, the user needs to interact with the system to examine the state of hardware devices and the application running on the chips. A common communication language using small packets to interact with each chip's MProc is required.

4.3. Boot-up protocol

The SpiNNaker multi-CMP system is configured in two phases. In the first phase, the processors run the Boot ROM code in batch mode to test processing cores and on-chip peripherals independently. In the second phase, the configuration process follows the SpiNNaker event-driven model to configure the whole system at run-time. The following sections explain the two phases.

4.3.1. Phase I – chip level configuration

Each SpiNNaker CMP must perform basic power-on testing and initialization based on the instructions in the Boot ROM. Initially the ARM968 Tightly Coupled Memory (TCM) is cleared and the processors access a high interrupt vector in Boot ROM at power-on reset. At this point, all the processors run at very low frequency (10 MHz) to conserve power. After testing the TCM, the bootstrap code is copied to the local instruction memory (ITCM) of each processor to enhance performance. The process of copying the code from the Boot ROM occurs simultaneously among the 18 processing cores, potentially causing some contention. However, once the code is in local memory, each processor can run it independently. To complete initial boot-up, each processor performs the following actions.

Each processor performs a power-on self-test (POST) on the processing core's peripherals, i.e. the communications controller, the vectored interrupt controller (VIC), the timer and the direct memory access (DMA) controller, to identify any fault. For every processing node to function properly, it is important that all its peripherals are functional. In case of a failure, the processing node logs the fault, then puts itself into a sleep mode to avoid any disruption to the CMP. The processors initialize their peripherals to a safe default setting. The communications controller is initialized to send and receive packets, the timer is initialized to generate an interrupt after every millisecond to synchronize system-wide activities, the VIC is configured to raise basic interrupts from devices, and the DMA controller is set up to perform read/write operations while flagging errors via events.

Unlike most CMP systems, the monitor processor is not hardwired at design time, as this introduces a single point of potential failure. In the SpiNNaker CMP, the monitor processor is selected from the 18 on-board processors at run time using instructions in the boot-up code. To this end, all healthy processors compete to access the System Controller across the System NoC. The System NoC's arbiter allows only one processor to access the System Controller, which selects the first processor to gain access as the monitor processor. The System Controller writes the ID of the selected monitor processor to an internal register, which is interrogated by all other processors in order to identify that monitor. All processors inform the System Controller of their state. For this purpose the System Controller includes a specific register. Each processor, after successful test and initialization of its peripherals, sets the bit corresponding to its ID in this register. This information provides the monitor processor with the state of other processors, and is used later for chip management, application configuration and chip status reporting to the Host PC (as explained later). Additionally, the System Controller maintains the state of other chip resources which can be queried from the outside autonomously.

At this stage all processors, except the monitor, go into sleep mode. The monitor processor, which is now managing the chip, performs chip-level testing and initialization. It switches the chip clock to a faster frequency, e.g. 200 MHz. Clock speed is controlled through the Phase-Locked-Loop (PLL) control register in the System Controller. It performs detailed device-level tests and writes the state of chip resources to the System Controller for later reporting to the external Host PC. It initializes the chip's global resources, such as the PL340 SDRAM Controller, the Router, and the System Controller. At this stage, the Router can route only NN packets, as the other three types of packet (MC, FR, and P2P) require relevant routing tables to be configured. It establishes if a PHY is present. If so, this chip may be connected to the Host PC. The monitor processor initializes the PHY, and if a live connection to the Host PC is detected, it initializes the Ethernet Interface to start receiving frames. If, however, the monitor processor does not detect a PHY, it disables the Ethernet Interface to save power. It configures the Watchdog Timer's interrupt and reset mechanism to reset the monitor processor, or the whole chip in case of a non-responsive monitor processor, as part of chip-level recovery. This mechanism will be explained later in Section 6.

After this process, the monitor processor also enters sleep mode, putting the chip in a listening state waiting for interrupt-signalled events, both internal (from Timer or Watchdog, etc.) and external (packets, Ethernet frames, etc.).

4.3.2. Phase II – system level configuration

In this phase, the configuration process runs as an event-driven application under the control of the VIC. At the chip(s) connected to the Host PC, the PHY generates connection-related events and the Ethernet Interface generates a "frame-received" event as an interrupt to the monitor processor. On all chips, the Monitor processor's communications controller generates a "packet-received" event when a message arrives from a neighbouring chip. Each interrupt triggers a different Interrupt Service Routine (ISR) to run the code for the related configuration job before putting the monitor processor to sleep again. These two different event-driven processes, i.e. packet received and Ethernet frame received, use separate communications protocols. The monitor processor on the Host-connected chip translates between the two protocols, converting the Ethernet frames it gets from the Host PC to packet-based messages, then issuing them either as broadcast or chip-specific NN

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

packets. In addition to these two interrupts, timer and DMA interrupts are also used in this process to support process time-out and memory transfer operations as part of the system-level configuration process (see Fig. 4).

After Phase I, each chip should receive a Hello message from all its neighbours within a certain time. If a given link times out, the monitor processor activates the "neighbour diagnostic" routine which establishes the problem and tries to cure the faulty chip (Section 6 will explain the process in more detail). Following any necessary time for the neighbour diagnostic process, the chips start executing event-driven system-level configuration. This, the main component of system-level boot-up, operates as follows.

The chip(s) connected to the Host PC notify system readiness to the Host PC by sending a "Hello" frame, indicating that the system has completed its Phase I process and subsequent diagnostics. The Host PC nominates one of the Host-connected chips to be the "reference chip" (i.e. the one to bear address (0,0)) and notifies it of the number of chips in the system. It then instructs the chips to assign unique chip addresses dynamically. The origin chip broadcasts its address (0,0) with the size of the system. Adjacent chips compute their addresses by size modulo addition to find a relative address (x,y) in a 2D SpiNNaker torus, and broadcast their address forward. This process continues outward to cover the whole system. Each chip configures its P2P table based on the logical location of the chips, using the size of the system as a reference, in order to perform default P2P routing. (The application can later modify these tables according to the system-level configuration). The Host PC requests the system state. Each chip reports its state to the origin chip using P2P packet(s). The origin chip accumulates these states and reports the result to the Host PC. The Host PC loads the run-time configuration code and application to the chips using a flood-fill mechanism as explained in Section 4.4. The Host PC computes the MC routing tables through a user interface application as per the user's application model, according to the chips' state, then configures the routing tables for each chip with the help of P2P packets as computed in the previous step. The Host PC instructs the monitor processors to notify application processors to load the application into their local memories and start running the application. The Host PC interacts with the system either to carry stimuli/responses on behalf of the application, or to interrogate the state of the system or application.

The protocol follows a set of instructions passed between the Host PC and the SpiNNaker system using Ethernet frames.

4.4. Application loading process

A typical SpiNNaker application will need to load neural support data into each SpiNNaker chip's memory before it can start its execution. The data may include the initial state of the application, neural network mapping, inter-neuron connectivity information, and synaptic information. In addition, a micro-kernel and utility functions library for the monitor processor may also need to be loaded into each chip. We need an efficient way to load this data in the minimum possible time.

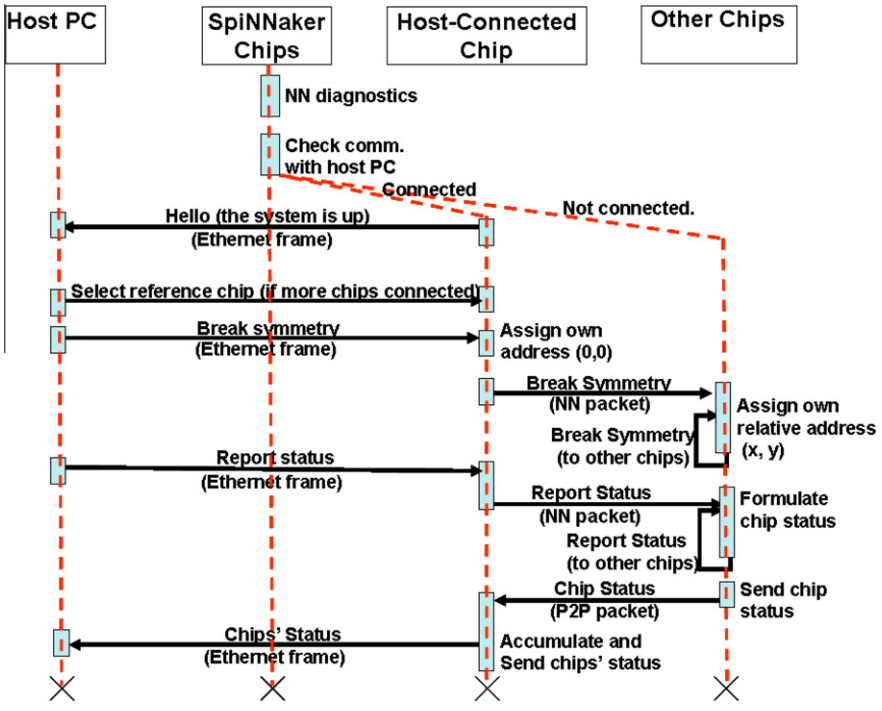


Fig. 4. Bootup process Phase II – system-level configuration.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

An efficient and fault-tolerant mechanism has been devised to load the application and data into the chips. During the inter-chip flood-fill process each CMP uses NN packets to send a 32-bit word of data at a time to its six neighbouring chips. The receiving chips store the data and broadcast it on to their neighbours. A pipelined "wave" of data thus flows from the Host-connected chip(s) to the whole system. To maintain data flow control, we have devised a custom instruction set to be used with NN packets. This protocol covers instructions to serialize the data, control the flow of data, request missing bits of data and various other configuration instructions. We have devised several flood-fill mechanisms such as "broadcast" or "selective forward multicast" which give a range of performance vs. robustness tradeoffs. Fig. 5 shows one such mechanism for flood-filling the data in the SpiNNaker system. Fig. 6 shows the sequence diagram of the flood-fill mechanism, which works as follows.

The Host PC loads the application and data as small ($\leqslant$1 K) data blocks to the system. It sends a data block along with its size, start address, and block-level Cyclic Redundancy Check (CRC) checksum to the Host-connected chip(s) using Ethernet frames. Each Host-connected chip performs a checksum test on the block to ensure its correct reception. If an error occurs, it requests the Host PC to resend the block. The chip(s) connected to the Host PC send an instruction through an NN packet to indicate the size and start address of the block to be transmitted. The system inserts an ID for each word into the routing key of the NN Packet to help with serialization and duplication control. Each neighbouring chip dedicates a memory space equivalent to the size of the data block being received. It also initializes a "word received" bitmap with a number of bits equal to the number of words in the block. On receipt of a packet, it will inspect the corresponding bit against the routing key (address) of the data in the packet. If a word has already been received, it will neither be stored in the memory buffer nor be transmitted further. Missing words (zeroes in the bitmap) can be requested from neighbours at the end of the flood fill. The last NN packet of data contains an instruction indicating the end of the block along with the block-level checksum in the payload. If the block passes the CRC test, the receiving chip's monitor processor loads it into the memory location specified at the start of flood-fill. The flood-fill process ensures that each chip receives the transmitted word at least twice during the flood-fill process, ensuring data delivery to each chip. At the end of the flood-fill process, the Host PC requests the state of each chip along with blocks received. At this stage, the chips can request missing blocks from each other or from the Host PC. During the "neighbouring chip recovery" process, chips surrounding the faulty one can send the data locally to the recovered chip. At the end of the application load process, the Host PC broadcasts an instruction to all the chips to start executing the application. The monitor processor on each chip sends a message to the application processors to start executing the application loaded to local instruction TCM.

5. Boot load verification and evaluation

To verify boot processing, we have developed a SystemC system-level model of a single- and multi-CMP SpiNNaker system. The model uses a cycle-accurate ARM968 instruction set simulator (ISS) from ARM SoC Designer together with

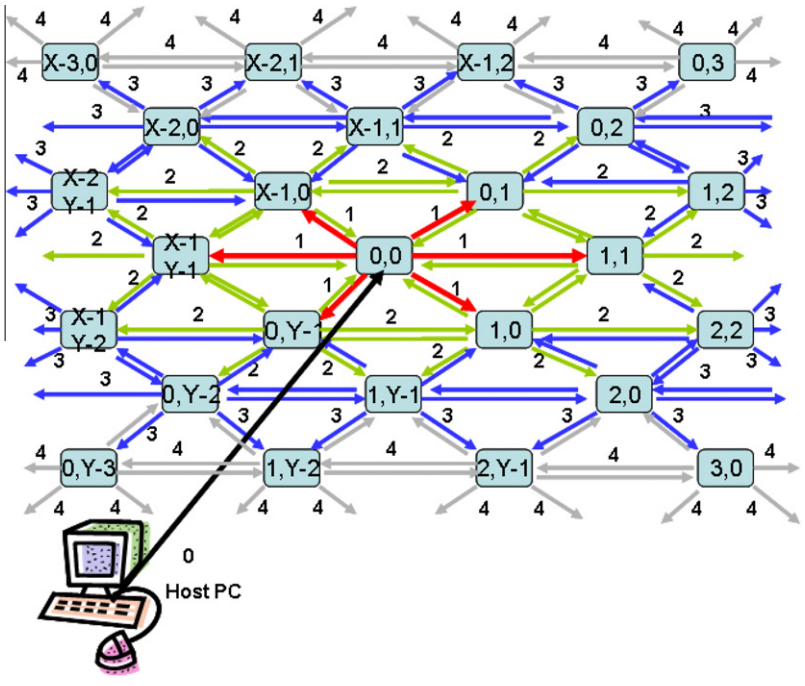


Fig. 5. Selective forward flood-fill.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

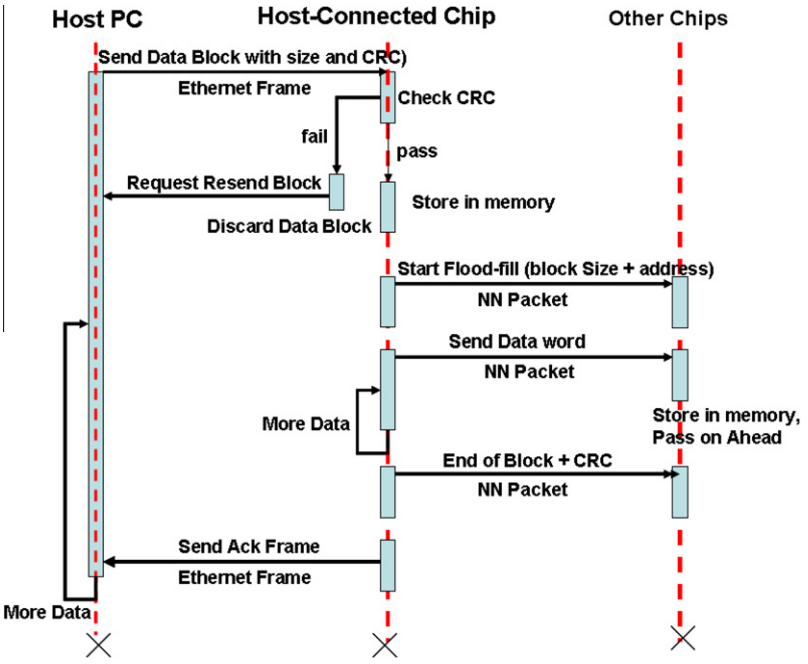


Fig. 6. Flood-fill process – sequence diagram.

Table 1
SpiNNaker CMP boot-up process Phase-I.

No. of CMPs	No. of procs. per CMP	CMP boot-up time (ms)	Sim. time (s)
1	1	1.37	6.00
1	3	1.37	8.28
5	3	1.37	41.60
9	3	1.37	77.86

cycle-accurate models for other ARM components such as the Interrupt Controller, SDRAM Controller, Watchdog Timer, AHB and APB busses, and memories. SystemC models for in-house designed components [17] have been added to make it a complete system model. Due to the combination of synchronous and asynchronous parts, the model as a whole exhibits cycle-approximate behaviour, as expected from SystemC Transaction Level Modelling (TLM) [12]. The system-level model for SpiNNaker simulates three processing cores for simplicity, enabling efficient simulation of a large system on a desktop PC (see Table 1, column 4, for the simulation performance on an Intel Core2 duo 1.6 GHz, 2 GB RAM running Windows XP). The model allows instruction- and cycle-level debugging of the hardware system and the application together, as one package. Simulation results have been verified with a Verilog top-level simulation of the system, providing cycle-accurate behaviour for detailed low-level signalling verification. The configuration process proposed in this paper has been implemented using ARM RealView Development Studio to generate a loadable Boot ROM binary image. Table 1 shows the chip-level boot-up time as a number of ARM968 CPU cycles (200 MHz). The boot-up time does not depend on the number of chips in the SpiNNaker system, since it runs concurrently on all the CMPs. Similarly, it does not depend on the number of CPUs in each CMP as the boot code is loaded to the local memory of each processing core before its execution. These results provide satisfactory verification of the design and functionality of the SpiNNaker system.

We also developed a high-level simulator for the SpiNNaker Communication Network to test the application-load process. We can simulate the SpiNNaker multi-CMP system to its full scale (64 K CMPs) using this simulation. The communication latency in this model is comparable with the SystemC model, as the network timings were those acquired from SystemC cycle-accurate simulation. We experimented with the following distribution algorithms for the application loading flood-fill process:

(a) *2Msg Forward (Fig. 7a)*: the monitor processor sends messages to the adjacent neighbours on the links diagonally opposite to the one it received the packet from, i.e. to the chips $(x + 1, y)$ and $(x, y + 1)$ in 2D torus configuration of the SpiNNaker multi-CMP system. This is the minimum number of packets required to perform an efficiently-pipelined flood-fill distribution of the application. Each packet needs 1 router cycle, i.e. a chip will use two router cycles to forward the message.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

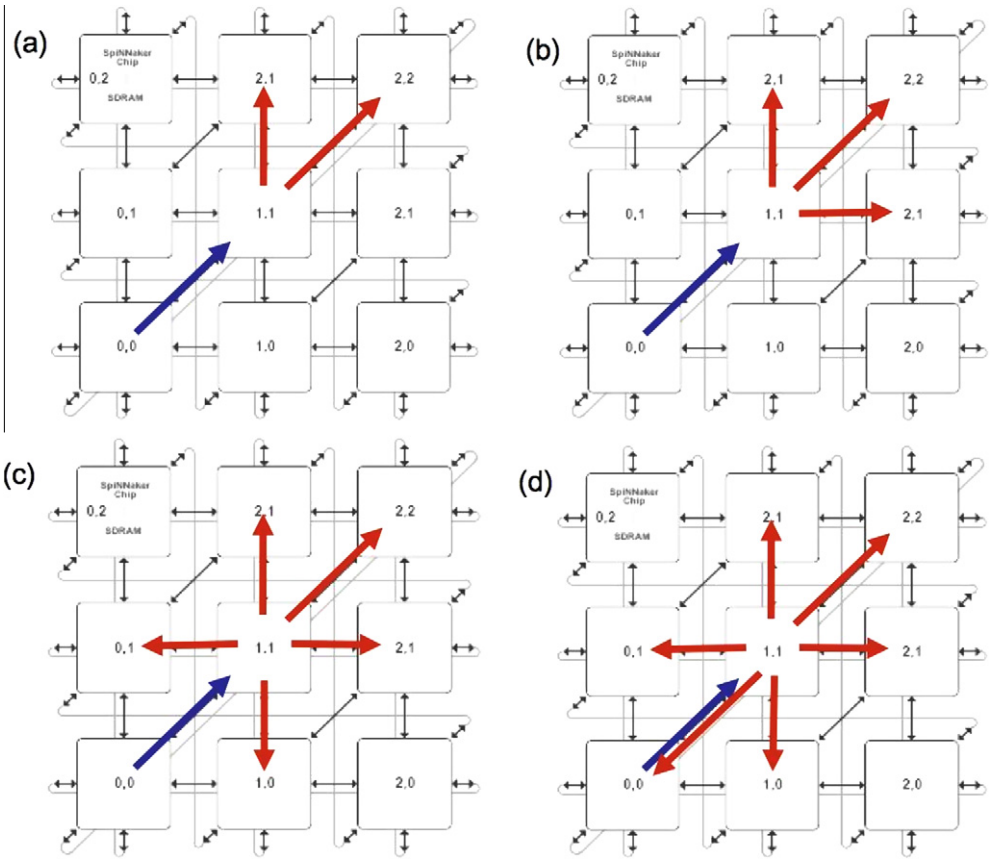


Fig. 7. Application loading process – flood-fill approaches.

(b) *3Msg Forward (Fig. 7b)*: this mechanism operates like 2Msg forward, except that the received packet is forwarded towards the diagonally opposite link and its two adjacent links. The process is slower than 2Msg Forward as it takes three router cycles to send packets to three neighbours, however, it is more reliable as it ensures that each chip receives a packet at least twice, which is not the case with the 2Msg Forward algorithm.

(c) *5Msg Forward (Fig. 7c)*: this process causes a CMP to send a copy of the packet to all neighbours except the one it received the packet from. This requires five router cycles as each neighbouring chip is sent the same packet individually.

(d) *Broadcast (Fig. 7d)*: the monitor processor on each chip uses the NN broadcast feature to send a copy of each packet to all its nearest neighbours using only 1 router cycle.

(e) *rndXX*: like 2Msg Forward but adds (XX%) probability to send packets to the other neighbours, i.e. a packet is forwarded to either 2, 3, 4, or 5 neighbouring chips selected randomly to avoid fixed inter-chip traffic congestion. We tested probability values 25% (rnd25), 50% (rnd50) and 75% (rnd75).

Besides testing the algorithm with all inter-CMP links intact, we experimented with various link failure models to evaluate fault-tolerance of our proposed flood-fill process. We simulated *vertical*, *horizontal* and *cross* link failure models with all the links along *X*-axis, *Y*-axis and both *X*, *Y*-axes, respectively, leading to networks partially split in vertical columns, horizontal rows and diagonal connections. We also tested the system with various numbers of random link failures from 1 K to 64 K (total 64 K $*$ 6 = 384 K links), at locations distributed uniformly.

We tested the system using 1, 2 and 4 Ethernet connections to the Host PC from the CMPs located at $(0,0)$, $(X/2,Y/2)$, $(X/2,0)$ and $(0,Y/2)$ where X and Y are the number of CMPs along X- and Y-axis, respectively, in the SpiNNaker 2D toroidal configuration (Fig. 1). Finally, we tested different network sizes, all of them square, ranging from 32×32 to 256×256.

Fig. 8 shows our results for error-free configurations with various application sizes to be loaded to the SpiNNaker system. The application load time depends linearly on the size of data; approximately 20 ms is needed to load an application equivalent to the size of a processing core's local memory (100 KB). Fig. 9 shows the impact of system size on the performance of various application loading mechanisms, while Fig. 10 shows the impact of the number of Ethernet connections. The results show that the application load time is essentially independent of both the number of Ethernet connections and system size in a large-scale multi-CMP system. The only relevant factors are the data size and the distribution policies. This is because of

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

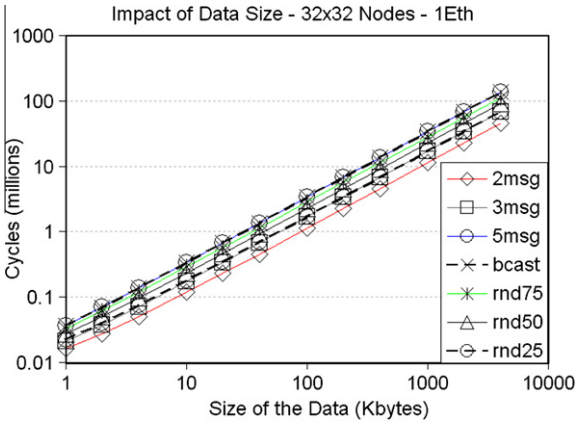


Fig. 8. Application loading on SpiNNaker multi-CMP system.

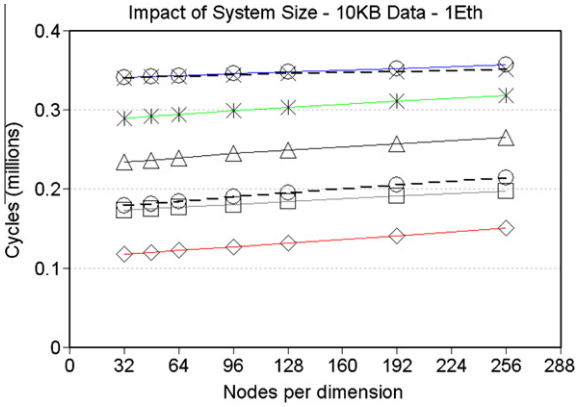


Fig. 9. Application loading on SpiNNaker multi-CMP system.

the perfect pipelining of the packets in the flood-fill process. Each chip passes packets on to its next neighbours, and in a 100 KB sample application the number of hops to reach the farthest point from the origin in the ($256 \times 256 = 64$ K chips) system is negligible compared to the number of (25 K) packets. As a result, the process rapidly transfers the application and data across the whole network. The flood-fill process is virtually independent of the number of Ethernet links to the Host, exhibiting negligible performance gain with four Ethernet links as compared to 1. There is, however, a considerable performance gain using the selective forward multicast flood-fill process (2msg or 3msg), since it relieves congestion. We found that the 2msg mechanism is the fastest, however, it does not ensure delivery of a packet at least twice to each CMP. The mechanism is not fault-tolerant as a broken link in the start of the flood-fill process may exclude all the chips in that direction from the application. It is necessary to ensure sufficient redundancy of transmitted packets so that a chip with blocked links should still get the ones from the duplicate link(s). Broadcast and 5msg techniques are the most fault-tolerant, however, these are the worst in performance due to network congestion caused by injecting too many packets. There is, therefore, a need to maintain a balance between performance and redundancy (fault-tolerance) which is achieved with 3msg and rnd25 mechanism.

Figs. 11 and 12 show the effect of the number of Ethernet connections on the application loading process in the presence of faulty inter-CMP links as explained above. Though connecting the Host PC to the SpiNNaker system at more than one point does not improve the application load time in a large-scale system, it does improve the fault-tolerance of the process, as a chip can receive a packet from various directions. This is particularly significant if the network is split in various regions due to link failures. Here, again, the broadcast mechanism proves to be the most robust, losing no packets in any failure mode, while the 3msg provides reasonably good fault-tolerance as it is not affected by horizontal and vertical failed links, or random link failures up to 8 K (a system degradation of about 2%).

6. Boot time fault recovery

The boot-time SpiNNaker configuration process has been devised to be as fault-resilient as possible, so as to bring the system into a functional state up to a certain degree of component-level malfunction. SpiNNaker hardware supports error

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

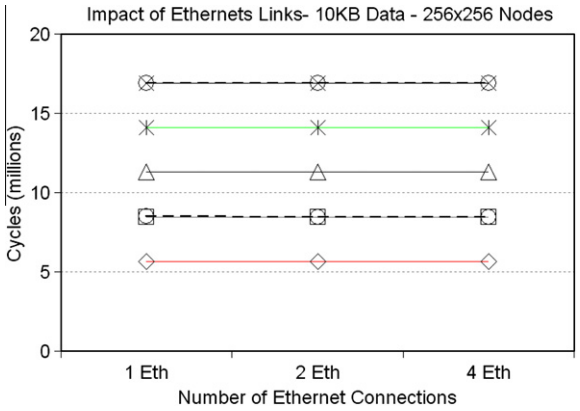


Fig. 10. Application loading on SpiNNaker multi-CMP system.

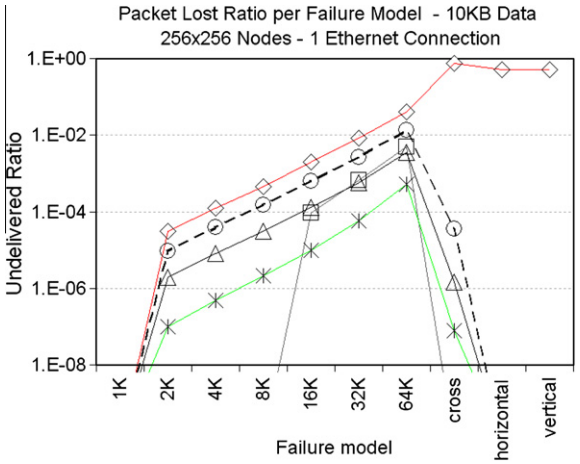


Fig. 11. Application loading on SpiNNaker multi-CMP system.

detection and handling for several classes of chip component through its configuration and management software. The MProc in each chip is responsible for dealing with such contingencies while the other processors continue to run the application. The SpiNNaker system provides resource redundancy at each level of its design to minimise single points of failure. Each chip's System Controller maintains a continuously updated state of all its processors and shared chip resources. Additionally, most chip components generate interrupts at the MProc to activate a relevant event-driven recovery routine for common faults. The processors, particularly the MProc, are loaded with fault-recovery routines to handle most of these exceptions.

SpiNNaker uses both the error-recovery techniques of replacement and reconfiguration to maintain its run-time functionality. The fault recovery process is autonomous and runs as an event-driven application (Section 3) as part of the SpiNNaker configuration process. The SpiNNaker configuration code for the processors, and particularly for the monitor processor, contains fault-recovery routines to handle certain envisaged faults. The monitor processor in each chip is responsible for dealing with chip-level fault handing and contributes to system-wide fault-tolerance with the help of the Host PC, while the other processors continue to run the application. The configuration process uses SpiNNaker's specially-designed fault-tolerance features to provide support for run-time local recovery, isolation of faulty components, and local reconfiguration, controlled by the monitor processor. The SpiNNaker configuration process makes use of the design features to provide a fault-tolerant hardware platform to the user. The following are some important fault-tolerance features of the SpiNNaker configuration process.

(a) *Monitor Processor Selection*. In contrast to a typical CMP system, where a monitor processor is hard-wired at design time (thus introducing a single point of failure), SpiNNaker chips do not have a dedicated monitor processor. The configuration process selects the monitor processor arbitrarily from the healthy processors for better fault-tolerance. The monitor processor can be replaced by any other healthy processor at run-time, in the case of a problem with the existing configured monitor processor.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

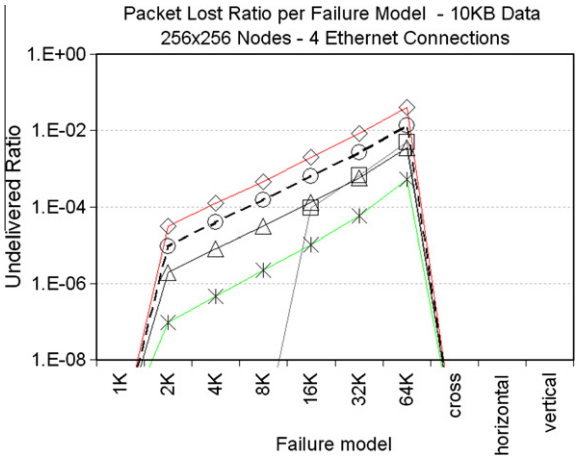


Fig. 12. Application loading on SpiNNaker multi-CMP system.

(b) *Boot ROM Failure*. In case of a Boot ROM failure, the configuration process makes use of a specially designed RAM-ROM address remapping feature in the SpiNNaker CMP to use System RAM for the boot-up process by copying the bootstrap code into the System RAM with the help of a neighbouring chip.

(c) *Chip-level Recovery*. To detect and recover a faulty SpiNNaker CMP, a detailed chip-level recovery mechanism has been devised which uses custom features in the chip. During chip-level initialization, the monitor processor tests all the devices and tries to reset any faulty components. This may allow a device to recover from a transient fault. If, however, a device does not respond after reset, the monitor processor records its state and informs the Host PC, while reporting the chip's state. Similarly, each processing node informs the System Controller of its state, which is recorded in a dedicated register. The monitor processor tries to bring a dead processing node up by resetting it, or notifies the Host PC. Chip-level management and recovery depends on the health of the monitor processor, which itself may be deadlocked due to a hardware or software fault. A Watchdog timer in each chip detects a non-responsive monitor processor and informs the System Controller to reset it or replace it with a healthy one. The System Controller has an option that will reset the monitor processor only on Watchdog interrupt, and contains a corresponding status register to record the reason for the last reset: power-on reset, Watchdog chip-level reset, or Watchdog soft reset only to the monitor processor.

(d) *NN Diagnostics and Recovery*. Using an NN packet, a chip can broadcast a packet to all its six neighbours simultaneously, can communicate with any of its chips individually, or "peek" and "poke" any of its neighbouring chip's resources. The peek and poke feature of the NN packet has been designed specifically to help bring up a dead chip during the configuration process. Using this feature, the chips surrounding a dead chip try to diagnose the reason by reading the information recorded in the dead chip's System Controller. Each chip tries to check its connections with six other chips by reading the neighbouring chips' status from their System Controllers using NN packets. If a chip does not get a response or the status indicates a problem in a chip, the six chips neighbouring the dead chip activate the "neighbour diagnostic" routine to recover the dead chip. One out of six healthy neighbouring chips is selected to cure the dead chip. The recovery routine tries to recover the chip from failures such as inter-chip link failure, non responsive monitor processor, or broken Boot ROM.

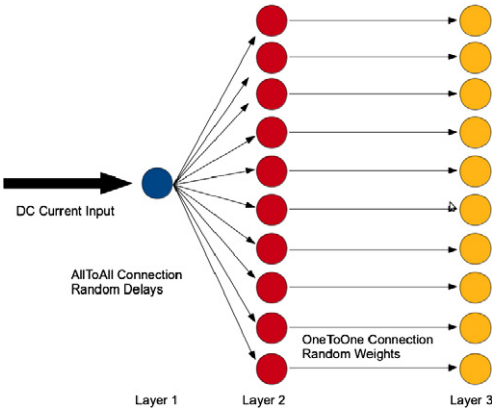
Table 2
NN diagnostic and recovery process.

S/No.	Fault	Measure taken	Time
1	Inter-chip link broken	Each Chip resets its non-responsive links The process is repeated a number of times after a delay of 3 ms before giving up	0.48 μs
2	Inter-chip link broken and could not be repaired	Disable link and communicate through a common neighbour. The non-responsive link was reset thrice at 3.9 ms, 6.1 ms and 8.9 ms after the boot-up. After three resets, the link was disabled and marked as dead in the ITCM to avoid sending packets to this port	6 ms
3	Monitor processor not responding	Option 1: (if the chip configuration has already been done by the previous monitor processor) the nurse chip changes the monitor processor ID to another processor, disables the current monitor processor and resets the newly selected processor to take over as the monitor processor	1.09 ms
4	Monitor process not responding	Option 2: (if the chip has not been initialized) the nurse chip resets all processors to restart the boot process	1.37 ms
5	Boot ROM broken	The nurse chip loads the Boot ROM image to the dead chip's System RAM, remaps RAM–ROM address and resets processors to boot from the System RAM	13.2 ms
6	Nurse chip failed to bring the dead chip up	disable all processors' clocks in the dead chip and report to the Host PC	1.2 μs

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

(e) *Connection to the Host PC*. In an integrated multi-CMP SpiNNaker system, we can configure more than one Ethernet link to connect the system to the Host PC to provide link redundancy and additional external I/O bandwidth (although they do not improve application load performance (Fig. 10)). Multiple Host PCs can also provide user-side redundancy. At boot time, each SpiNNaker chip with a PHY is unaware of the identity of its Host PC. The chip determines the Host IP/MAC addresses from the responses it receives to its own periodic "Hello" packets. Administrators can define which chips are allocated to which Host PC links. To provide the redundancy function, Host PCs monitor each another and can dynamically substitute for a failed Host PC, using Multicast MAC/IP addressing on the external Ethernet network.

(f) *Fault-Tolerance in Application Loading*. The application loader uses broadcast (or multicast) to flood-fill the application and its data into the chips. As a consequence of the flood-fill process, each chip receives redundant data, maximising the probability of delivery of all the data and its associated application without retransmission. Without redundancy, a




```
from pyNN.spiNNaker import *                          # Imports the pyNN.spiNNaker module
from pyNN.random import NumpyRNG, RandomDistribution # Import Random Modules
kernelseed  = 43210987                                # random seed
rng = NumpyRNG(kernelseed)                            # Numpy RNG
# Simulation Setup
dt = 1                                                # time step [ms]
delay = 10                                            # max delay
setup(timestep=dt, max_delay=delay)                   # Add SpiNNaker parameters
# Set SpiNNaker Parameters
runTime(500)              # Simulation time
setChipsNumber(1,1)
N = 5                                          # Size of the population
cell_params_izh = {'v_in'        : -80,        # Initial V
                   'u_in'        : 0,          # Initial U
                   'a'           : 0.02,
                   'b'           : 0.2,
                   'c'           : -65,
                   'd'           : 8,
                   'i_offset'    : 0           # Bias current
                   }
# Layer 1 Population: 1 Izhikevich neuron
Layer1 = Population(1,              # size
                    Izhikevich,     # Neuron Type
                    cell_params_izh, # Neuron Parameters
                    label='Layer1'
                    )     # Label
# Layer 2 Population: 10 Izhikevich neurons
Layer2 = Population(10,             # size
                    Izhikevich,     # Neuron Type
                    cell_params_izh, # Neuron Parameters
                    label='Layer2'
                    )     # Label
# Layer 3 Population: 10 Izhikevich neurons
Layer3 = Population(10,             # size
                    Izhikevich,     # Neuron Type
                    cell_params_izh,
                    label='Layer3'   # Neuron Parameters
                    )     # Label
# DC Current injected into layer 1 from 100 to 400 ms
cs = DCSource(amplitude=20, start=100, stop=400)
cs.inject_into(Layer1)                    # Input to all neurons in i1
# Connection from Layer 1 to Layer 2
l1_l2 = Projection( Layer1,               # Source
                    Layer2,               # Destination
                    AllToAllConnector()
                    )
# Connection Parameters: Random delay between 1 and 10
uniformDistr = RandomDistribution('uniform', [1,10], rng)
l1_l2.setWeights(20)                      # Weight = 20
l1_l2.randomizeDelays(uniformDistr)       # Uniform RNG between 1 and 10
# Connection Layer2 -> Layer3
l2_l3 = Projection( Layer2,               # Source
                    Layer3,               # Destination
                    OneToOneConnector()   # Layer2 -> Layer 3
                    )
l2_l3.randomizeWeights(uniformDistr)      # Random Weights between 1 and 10
l2_l3.setDelays(1)                        # Uniform Delays (1 ms)
convertNetwork2SpiNNaker(vars().items()) # Generates files for InitLoad
```

Fig. 13. Simple neural network and PyNN script.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

chip expecting to receive packets only from a single link that happened to be broken would remain unpopulated with data, without which it could pass no data onward, leaving all the chips in its transmission path unconfigured. Redundant packets can also be used to verify the data by comparing matching packets. While each chip receives the same executable code, the monitor processor configures the application according to its chip location. Since each chip contains the same source, any chip can be reconfigured at run-time to take on the role of any other chip, and can likewise provide missing data locally to a chip that recovers from an initial faulty state.

Using the SystemC simulation, we tested the NN Diagnostics and Recovery process to recover a partially dead chip from a transient failure. The results in Table 2 show the action taken as part of the recovery routine and the time taken to execute the solution.

7. Implications for application development support

As has previously been stated, SpiNNaker's primary purpose is to run large-scale neural simulations. Within this context, the rôle of the boot system is twofold: to bring the chips into a working state, and then to load the *configuration*, the specific neural network model and topology that will support the application to be run. While the first phase is automatic, the second phase requires configuration files from the user to specify the model. Since we expect a significant proportion of SpiNNaker users will be neurobiologists and psychologists with minimal knowledge of, or even interest in, the implementation details of the hardware, it is important to abstract these files using automated translation tools that convert a high-level specification into the low-level binary files the configuration loader (Section 4.4) requires. We have created a 3-level software configuration hierarchy that allows neural modellers to describe a model using standard neural network modelling applications and use an HDL-like tool flow to generate the boot files. At the highest level of abstraction: Model Level, the system uses PyNN, a general-purpose neural modelling interface that translates descriptions into a number of different "back-end" simulators. We have extended PyNN with a SpiNNaker back-end module. This tool converts to an intermediate level of abstraction: System Level, that represents SpiNNaker-specific models and functions without explicit reference to device details. A second translator converts this intermediate level into the boot files representing the lowest level of abstraction: Device Level. The boot process works directly with Device Level files, which represent actual execution binaries such as device drivers and internal data structures.

As a specific example, consider a simple network composed of 21 Izhikevich-model neurons connected in a basic 1-10-10 structure (Fig. 13). At the top level the user creates the network description in PyNN (code sample 13). PyNN then converts this to a set of mapping files (Figs. 14a and b). The user then runs the intermediate mapper "initload" to convert them into data binaries. The boot process now loads these binaries, using an internal program to place files for the DTCM, ITCM, SDRAM, and router, respectively, into the specific addresses a configuration script specifies. In the case of this simple network, the system can map all the neurons to a single processor, requiring a single routing table entry. Because the boot process automates the loading of these configuration files, the user can specify the model in PyNN without having to consider how it will be mapped to the hardware; for example, which processors will contain which neurons. Meanwhile, the flood-fill process distributes the configuration files to their target processors without having to consider the topology of the model.

We are also developing the Host PC and Systems in order to configure, load and interact with the application running on the system as well as manage the equipment. The Host PC provides a consolidated view of the underlying hardware

```
ID 0: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 1: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 2: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 3: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 4: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 5: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 6: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 7: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 8: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 9: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 10: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 11: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 12: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 13: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 14: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 15: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 16: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 17: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 18: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 19: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
ID 20: 1; IZK; -80, 0, 0.02, 0.20, -65, 8, 0
```

Fig. 14a. SpiNNaker system level neurons file.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

```
ID 0
: 1, 20.00, 0, 4
: 2, 20.00, 0, 3
: 3, 20.00, 0, 8
: 4, 20.00, 0, 4
: 5, 20.00, 0, 8
: 6, 20.00, 0, 1
: 7, 20.00, 0, 1
: 8, 20.00, 0, 7
: 9, 20.00, 0, 8
: 10, 20.00, 0, 6
ID 1
: 11, 8.11, 0, 1
ID 2
: 12, 6.55, 0, 1
ID 3
: 13, 6.36, 0, 1
ID 4
: 14, 4.43, 0, 1
ID 5
: 15, 8.57, 0, 1
ID 6
: 16, 1.15, 0, 1
ID 7
: 17, 6.67, 0, 1
ID 8
: 18, 4.73, 0, 1
ID 9
: 19, 9.92, 0, 1
ID 10
: 20, 5.77, 0, 1
```

Fig. 14b. SpiNNaker system level connections file.

reliability and performance via a graphical user interface (GUI), providing facilities for system level diagnostics and debugging, graphical visualisation of the system state, and processor and communication network loading. The users will have the ability to interact at a user-definable level – from whole-system view, down through chip, and core, to individual neuronal activity levels, all in real-time. An underlying lightweight management framework is being developed which provides data to bespoke applications as well as to generic management platforms via SNMP. The Network Management System may scale from a single Host PC (e.g. Fig. 1), to a larger deployment with multiple management stations providing differing management functions and interfaces for users, and network and "box-level" hardware resilience – including geographic dispersal. User-level functions (being developed by the SpiNNaker Application Group at the University of Southampton) to configure the application and map the neural connectivity before loading it into the SpiNNaker system will use this interface before loading the application. A sample application will also be provided with the interface as a tutorial for running spiking neural applications on SpiNNaker.

8. Conclusions

We have developed an infrastructure for configuring user-defined applications on a chip multiprocessor optimised for general-purpose neural network simulation with the following components:

(a) An efficient fault-tolerant configuration mechanism.
(b) An efficient flood-fill application load process.
(c) A library of device routines that abstracts functionality to the model level.
(d) A cycle-accurate SystemC model for the SpiNNaker system and a high-level communication simulation.

There remains considerable work on high-level user components. We are currently developing Management System user interfaces that provide access to the low-level tools we have already developed, and to additional management functions for both hardware and software. Ongoing research is also investigating methods for dynamic and potentially autonomous system reconfiguration at run time, by using data collected by the management system, allowing for neural developmental processes. In addition, further work is necessary in identifying efficient routing and mapping schemes. Work is ongoing on developing the library functions and high-level descriptions to support multiple classes of neural network in a general-purpose library of neural functionality that allows the user to specify a model at a high level and automatically instantiate library files to generate the requisite routines and mappings, a process akin to hardware synthesis.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003

SpiNNaker represents a fundamentally new architecture for neural networks: a "neuromimetic" chip. As such it realises the FPGA's reconfigurability and ability to model any neural network, and the scalability and performance of traditional hardwired neural ASICs. We have shown that this design approach makes it possible to develop a system that leverages specific performance gains from application-focussed circuitry without constraining model choice or size. Nonetheless, such a "blank slate" approach requires novel methods for configuration and execution, or the modeller may be faced with a chip so general that getting it to do anything useful is a research project in itself. Our research has therefore created an essential infrastructure the modeller needs to make SpiNNaker a useful practical tool for hardware neural modelling. Considered as a general parallel architecture, SpiNNaker offers an alternative to the traditional large-scale parallel machine: instead of developing a completely general-purpose chip and designing the application to match the hardware capabilities, we designed a chip matched to the needs of a specific application known to be highly parallel, and provided general-purpose software tools to develop applications. This represents a path for parallel processing akin to an embedded system, where it is understood at the outset that it is running a definable single application. If parallel processing is most effective with specific parallelisable tasks, it seems more logical to develop task-optimised parallel devices to complement general-purpose uniprocessors than to try to create a parallel processor to replace the uniprocessor outright.

Acknowledgements

The SpiNNaker project is supported by EPSRC (Grant EP/D07908X/1), ARM and Silistix. S.B. Furber holds a Royal Society-Wolfson Research Merit Award. J. Navaridas is supported by a doctoral grant of the UPV/EHU and by the Spanish Ministry of Education and Science (Grant TIN2007-68023-C02-02).

References

[1] N.R. Adiga, M.A. Blumrich, D. Chen, et al, Blue Gene/L torus interconnection network, IBM Journal of Research and Development 49 (2005) 289–301.
[2] S.R. Alam et al, An evaluation of the Oak Ridge National Laboratory Cray XT3, International Journal of High Performance Computing Applications archive 22 (2008) 52–80.
[3] S.R. Alam, J.A. Kuehn, R.F. Barrett, J.M. Larkin, M.R. Fahey, R. Sankaran, P.H. Worley, Cray XT4: an early evaluation for petascale scientific simulation, in: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07), November 2007.
[4] G. Almasi, L. Bachega, R. Bellofatto, J. Brunheroto, C. Caçscaval, J. Castaños, P. Crumley, C. Erway, J. Gagliano, D. Lieber, P. Mindlin, J.E. Moreira, R.K. Sahoo, A. Sanomiya, E. Schenfeld, R. Swetz, Sytem management in the BlueGene/L supercomputer, in: Proceedings of the 2003 International Parallel and Distributed Processing Symposium (IPDPS'03), IEEE Press, 2003.
[5] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, B. Verghese, Piranha: a scalable architecture Based on single-chip multiprocessing, in: Proceedings of the 27th International Symposium on Computer Architecture, 2000, pp. 282–293.
[6] T. Binzegger, R.J. Douglas, K.A.C. Martin, A quantitative map of the circuitry of cat primary visual cortex, Journal of Neuroscience 24 (39) (2004) 8441–8453.
[7] Cray Inc., Cray X1E Datasheet, January 2005a. <http://ed-thelen.org/comp-hist/CRAY-1-HardRefMan/CRAY-1-HRM.html>.
[8] Cray Inc., Cray XT3 Datasheet. Cray Inc., January 2005b. <http://www.craysupercomputers.com/downloads/CrayXT3/CrayXT3/Datasheet.pdf>.
[9] P. Dayan, L. Abbott, Theoretical Neuroscience, MIT Press, Cambridge, 2001.
[10] S.M. Diesburg, P.A. Gray, High performance computing environments without the fuss: the bootable cluster CD, in: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05), IEEE Press, 2005.
[11] S. Furber, S. Temple, A. 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.
[12] F. Ghenassia, Transaction-level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.
[13] L. Hammond, B.A. Nayfeh, K. Olukotun, A single chip multiprocessor, IEEE Computer Magazine 30 (April) (1997) 79–85.
[14] R.A. Haring, R. Bellofatto, et al, Blue Gene/L compute chip: control, test and bring up infrastructure, IBM Journal of Research and Development 49 (2/3) (2005) 289–301.
[15] A. Jahnke, T. Schönauer, U. Roth, K. Mohraz, H. Klar, Simulation of spiking neural networks on different hardware platforms, in: Proceedings 1997 International Conference Artificial Neural Networks (ICANN 1997), 1997, pp. 1187–1192.
[16] X. Jin, S. Furber, J. Woods, Efficient modelling of spiking neural networks on a scalable chip multiprocessor, in: Proceedings of 2008 International Joint Conference on Neural Networks (IJCNN2008), 2008.
[17] M. Khan, X. Jin, S. Furber, L. Plana, System-level model for a GALS massively parallel multiprocessor, in: Proceedings of the 19th UK Asynchronous Forum, London, Sep. 2007, pp. 9 – 12.
[18] M. Khan, D. Lester, L. Plana, A. Rast, X. Jin, E. Painkras, S. Furber, SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor, in: Proceedings of the 2008 International Joint Conference on Neural Networks (IJCNN2008), 2008.
[19] H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.
[20] W. Maass, C.M. Bishop, Pulsed Neural Networks, MIT Press, Cambridge, Massachusetts, 1998.
[21] R.W. Means, Simulation of spiking neural networks on different hardware platforms, in: Proceedings of the 1994 IEEE International Conference Artificial Neural Networks, IEEE Press, 1994, pp. 10–16.
[22] A. Rast, S. Yang, M. Khan, S. Furber, Virtual synaptic interconnect using an asynchronous network-on-chip, in: Proceedings of 2008 International Joint Conference on Neural Networks (IJCNN2008), 2008.
[23] U. Seiffert, Artificial neural networks on massively parallel computer hardware, Neurocomputing 57 (3) (2004) 135–150.
[24] T.P. Trappenberg, Fundamentals of Computational Neuroscience, Oxford University Press, New York, 2002.

Please cite this article in press as: M.M. Khan et al., Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Comput. (2011), doi:10.1016/j.parco.2011.02.003