Parallel Computing 39 (2013) 693–708

Contents lists available at ScienceDirect

Parallel Computing

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

# SpiNNaker: Fault tolerance in a power- and area- constrained large-scale neuromimetic architecture

Javier Navaridas [a,*], Steve Furber [a], Jim Garside [a], Xin Jin [b], Mukaram Khan [c], David Lester [a], Mikel Luján [a], José Miguel-Alonso [d], Eustace Painkras [a], Cameron Patterson [a], Luis A. Plana [a], Alexander Rast [a], Dominic Richards [a], Yebin Shi [b], Steve Temple [a], Jian Wu [e], Shufan Yang [f]

[a] University of Manchester, Manchester, United Kingdom
[b] ARM Ltd., Cambridge, United Kingdom
[c] National University of Sciences and Technology, Islamabad, Pakistan
[d] University of the Basque Country, San Sebastian, Spain
[e] University of Utah, Salt Lake City, UT, USA
[f] University of Ulster, Belfast, United Kingdom

ARTICLE INFO

*Article history:*
Received 19 December 2011
Received in revised form 2 May 2013
Accepted 4 September 2013
Available online 13 September 2013

*Keywords:*
Fault tolerance
Globally asynchronous locally synchronous
Low power system
Massively-parallel architecture
Spiking neural networks
System-on-chip

ABSTRACT

SpiNNaker is a biologically-inspired massively-parallel computer designed to model up to a billion spiking neurons in real-time. A full-fledged implementation of a SpiNNaker system will comprise more than $10^5$ integrated circuits (half of which are SDRAMs and half multi-core systems-on-chip). Given this scale, it is unavoidable that some components fail and, in consequence, fault-tolerance is a foundation of the system design. Although the target application can tolerate a certain, low level of failures, important efforts have been devoted to incorporate different techniques for fault tolerance. This paper is devoted to discussing how hardware and software mechanisms collaborate to make SpiNNaker operate properly even in the very likely scenario of component failures and how it can tolerate system-degradation levels well above those expected.

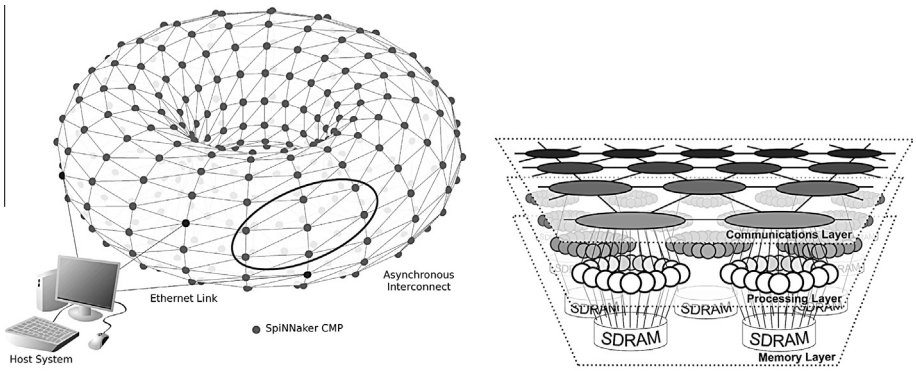


## 1. Introduction

SpiNNaker is an application specific design intended to model large biological neural networks – the name "SpiNNaker" being derived from 'Spiking Neural Network architecture'. It consists of a toroidal arrangement of processing nodes, each incorporating a purpose-built, multi-core System-on-Chip (SoC) and an SDRAM memory (Fig. 1). Neurons are modelled in software running on embedded ARM968 processors; each core is intended to model a nominal 1000 neurons. Small-scale SpiNNaker systems have successfully been used as control systems in embedded applications [1], providing robots with real-time *stimulus-response* behaviour as described in [2]. However the ultimate aiming of the project is to construct a machine able to simulate up to $10^9$ neurons in real time. To put this number in context some small primates have brains with slightly lower neuron counts whereas the human brain has roughly 86 times this number [3]. To reach this number of

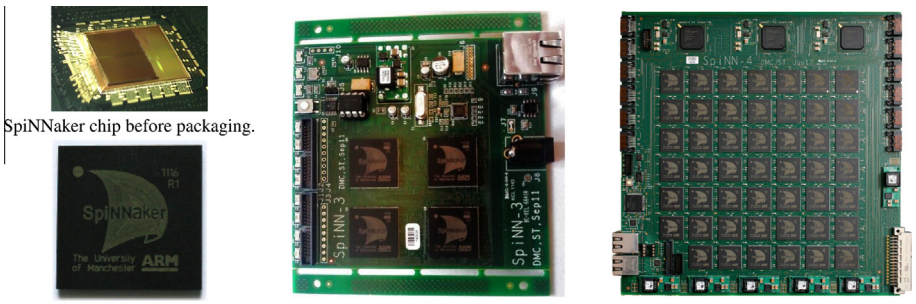
* Corresponding author. Tel.: +44 (0) 161 275 6143; fax: +44 (0) 161 275 6204.
*E-mail address:* javier.navaridas@manchester.ac.uk (J. Navaridas).

0167-8191 
http://dx.doi.org/10.1016/j.parco.2013.09.001




(a) Global view (b) Close-up view

**Fig. 1.** Overall view of SpiNNaker.




Packaged SpiNNaker chip. 4-chip developmental board. 48-chip production board

**Fig. 2.** SpiNNaker chip and boards.

neurons more than one hundred thousand integrated circuits will be needed (half of which are SpiNNaker chips and the other half SDRAMs).

A system of this scale may be expected to suffer component failures and many features of its design are included to provide a certain degree of fault tolerance. These features can sometimes be justified on cost alone: the overall yield for the 100 $mm^2$ SpiNNaker SoC was estimated, using public domain yield statistics on a 20-core, at 50% fault-free chips, 25% single-fault chips, 10% two-fault chips and the remaining 15% will be unusable due to critical failures. Early test on the production chip (in Fig. 2) show similar, if rather better, yield characteristics (see Section 4). The 35% of chips having one or two faults would not be usable without fault tolerance features. Fault tolerance is addressed at a number of levels, not least the application itself, which is intrinsically fault-tolerant. SpiNNaker incorporates measures to enable continued function in the presence of faults; in fact it has been designed as a power- and cost-effective fault-tolerant platform.

The major defence against faults in such a system is the massive processing resource. Processors are almost free and dedicating a small proportion of the processing power for system management and reconfiguration yields significant distributed 'intelligence' without much impact on the application. From the outset the intention has been to allocate one core on each SoC entirely to system management; if this eventually proves insufficient it is simple to delegate a second core to this task. Cores devoted to system management can identify and map around failed devices at run time.

Particular attention has been paid to inter-chip communications where link failures or transient congestion may be routed around rapidly without software intervention. Finally some more conventional techniques – such as automatic CRC generation and checking and watchdog timers – are employed in each processing node. As a large-scale system has not yet been built the full possibilities of software reconfiguration have yet to be explored. However statistical models of the architecture have been developed and used to verify the principles, and the hardware mechanisms themselves have been tested in silicon in small-scale (4 chip) systems. The construction of a larger machine is in progress.

## 2. Background

This section reviews common terminology on fault-tolerance and microelectronics, introducing several important concepts related to SpiNNaker and putting in context how fault tolerance is addressed.

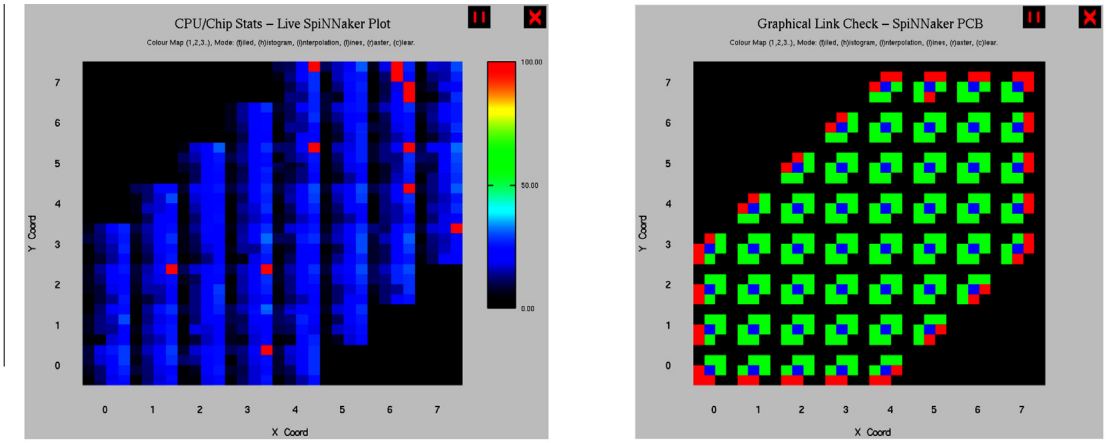
Throughout this paper, we differentiate between soft and hard errors. *Soft errors* are transient errors – usually produced by electromagnetic noise – that affect the state of a bit to an extent that it swaps its value (from 0 to 1, or vice versa). Cosmic rays are nowadays the main cause of soft errors [4]. In contrast, *hard errors* are permanent errors due to physical defects, usually introduced during fabrication. Some authors consider a third type of error, *intermittent* failures in which a component is barely stable and behaves irregularly as correct or as erroneous, the main triggers for one behaviour or the other being environmental factors (such as temperature, or voltage) [5]. We consider intermittent failures as hard failures and deactivate components that exhibit this behaviour.

All units within a SpiNNaker chip are provided with two levels of reset. A 'soft' reset is a signal to the state machines to abandon their operation *at the next convenient opportunity*, thus allowing any handshakes to complete first. The 'hard' reset involves *switching off* a component and restarting it in order to reach its initial state. Note that the latter is really intended only for power-up.

*Globally Asynchronous Locally Synchronous* (GALS) technology offers the possibility of synchronous and asynchronous logic to coexist, obtaining the best of each world [6]. Most devices use synchronous logic whereas communication between them is implemented using asynchronous fabrics. GALS simplifies development and reduces power consumption but, in contrast, makes fault tolerance difficult due to the lack of time awareness.

The three main elements for fault-tolerance are the Host, the System Controller and the Monitor Processor and Process. The *Host* is a regular computer which runs an application that interfaces with SpiNNaker giving the Host a range of control operations over the hardware. The Host is in charge of starting the system, uploading neural applications and data and looking after the status of the system once it starts its execution. It includes a User Interface that allows exploration of the status of SpiNNaker components (see Fig. 3). The *Monitor Process* is the application in charge of controlling the status of each chip components. It requires a dedicated core, namely the *Monitor Processor*, which is selected during the boot-up process from all the functional cores. The Monitor Process uses the *System Controller*, a specialized piece of hardware, to detect and try to heal failing components. The System Controller supports soft and hard resets of the different components within a chip and also communicates with the System Controller in neighbouring chips.

Watchdog devices are added to the design in order to supervise the correct operation of critical components such as the Monitor Process, or the communication ports. If a component does not respond for a predetermined amount of time, the watchdog will apply 'soft' reset first, only resorting to 'hard' reset if this fails. If both resets fail the watchdog will mark the component as faulty in the System Controller so that the Monitor Process can switch it off or, alternatively, try more elaborated nursing.




(a) User Interface showing the status of the cores. The colour gradient represents the load of each node's core. Red represents a core failure. 16 worker cores belonging to a chip are arranged in a 4x4 grid. Most cores have low load black/blue tones (< 25%). In this example there are 8 chips which have a 1-core failure and, in addition, the chip in 6,7 has 4 core failures.

(b) User Interface showing the status of the links. Green means the link is working. Red means the link is faulty or disconnected. All the links in the periphery of the board are disconnected. The southwards link in chip 5,7 has a failure, the chip cannot receive from its neighbour in that direction. However as links are unidirectional, it can send packets in that direction (note that northwards link in 5,6 is green).

**Fig. 3.** User Interface showing the status of the chips in a 48-chip board. The hexagonal shape represents the actual connection of the chips in a 48-chip board.

**Table 1**
Relative area of the different components within a SpiNNaker chip. See Fig. 4(a) for the actual overlay.

| Component | Relative area (%) |
|---|---|
| Tightly coupled memories (×18) | 50 |
| ARM cores (×18) | 29 |
| System NOC | 11 |
| Router | 6.7 |
| System SRAM memory | 0.96 |
| SDRAM interface | 0.75 |
| Phase-locked loops (×2) | 0.67 |
| Ethernet interface | 0.65 |
| ROM memory | 0.25 |

SpiNNaker's fault tolerance relies mainly on redundancy: 18 cores, 6 output links, 2 PLLs (phase locked loops) and the memory subsystem. The main strength of this redundancy is that components do not have their identifiers hard-coded, and therefore the functionality of one component can be covered seamlessly by any redundant one. Practically, this means that critical components such as the Monitor Processor are extremely reliable.

Table 1 shows the relative areas of the different components of the chip to put in context their likelihood of fail. The largest part of the chip is devoted to cores and TCMs, the most redundant and therefore less critical components of SpiNNaker.

## 3. Overview of SpiNNaker

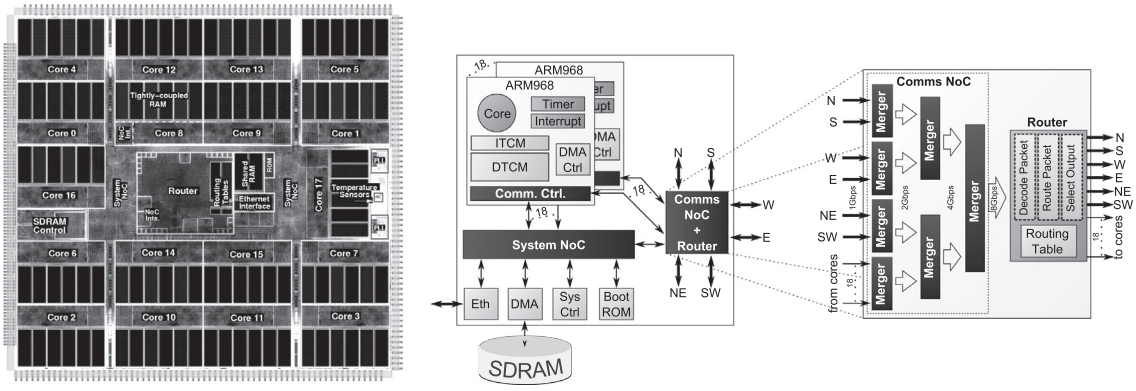
### 3.1. Application-induced architecture

SpiNNaker simulates spiking neural networks using Izhikevich [7] and Leaky Integrate and Fire [8] models which emulate the dynamics of biological neural systems. However SpiNNaker has an architecture general enough to run other flavours of application [9]. For example it also supports Multilayer Perceptron models [10] and other non-neural applications such as ray-tracing, many body interaction, finite element analysis and analogue circuit simulation.

Spiking neural systems have abundant parallelism and no explicit requirement of coherence as only local information is used by the neurons. The process is as follows: each neuron has a membrane potential which is affected by incoming stimuli (signals). If the membrane potential exceeds a given threshold, the neuron discharges and fires a signal (a so-called *spike*) which is transmitted to all neurons connected through a synaptic connection, typically in the order of $10^3$ [11]. Biological neurons work in a noisy environment [12] and, indeed, die during normal operation (adult humans lose about one neuron per second [13]). Thus their operation is neither perfect nor deterministic.

The SpiNNaker architecture reflects this behaviour. Neurons are modelled as event-driven applications executed by the processing cores. Spikes are represented by short network packets (40 bits) using Address-Event Representation (AER), a format widely used in neural network models [14–16]. Packets are *multicast* routed in hardware with the on-chip routers replicating them as necessary to reach all their destinations.

Given that digital electronics are orders of magnitude faster than the biological process – for example, biological spikes are propagated through an axon for up to 20 ms while transmitting a packet through the SpiNNaker interconnection network




(a) Chip overlay (b) Schematic design

**Fig. 4.** Architecture of a SpiNNaker chip.

should take a few *microseconds* at most – it is possible to multiplex many neurons onto a processor and many spikes onto a network although the consequence of a single-point failure can easily be more serious than the loss of one neuron.

### 3.2. SpiNNaker chip

The basic unit of the system is the SpiNNaker chip (Fig. 4), custom GALS SoC with a network router and 18 cores sharing some resources such as a SRAM, a boot ROM, a System Controller, an Ethernet interface and an 128Mbyte off-chip (but in-package, see Fig. 2) SDRAM.

Each *processing core* is an ARM968, with two *private* tightly coupled memories for instructions (ITCM) and data (DTCM), some peripherals – including direct access to the Comms NoC – and a bridge to the shared resources.

## 4. Fault tolerant architecture

From Section 3 some fault-tolerant features will already be apparent. Firstly the application itself is robust against minor perturbations in timing and should tolerate a percentage of missing spikes and neurons. Secondly, there is a huge hardware resource available which provides a high degree of redundancy.

Although each node has 18 cores, the intended use does not *require* any specific number of cores, merely (any) one to provide node control and *some* others to run the application. Indeed for cost purposes it is intended to use some flawed devices; yield estimates suggest that this may improve the usability of manufactured dice from 50% to around 80%. Based on the area use of the die the majority of flaws may be expected to be in local memories; these may leave a core degraded but still usable although the simplest action is still to shut it down.

Preliminary evidence from the first batch of fabricated chips suggests these estimates to be appropriate if slightly pessimistic. Of 46 chips, 30 (65%) were flawless chips, 12 (26%) have 17 working cores and 4 (9%) have more serious problems. Of the 12, 11 have private memory faults and one a peripheral logic fault. From this small sample it seems likely that 42 of 46 (93%) dice will be serviceable because manufacturing faults can be tolerated. This represents an increase of roughly 40% in terms of achievable computing power (from 540 to 744 cores).

This redundancy can also be used to protect against (less likely) run-time faults by offloading work. By keeping a stand-by core on each node a run-time fault can be accommodated without too much effort, particularly as the majority of the data is held in the separate, shared SDRAM. The SDRAM devices are 'known good' before packaging. Each provides a node with more

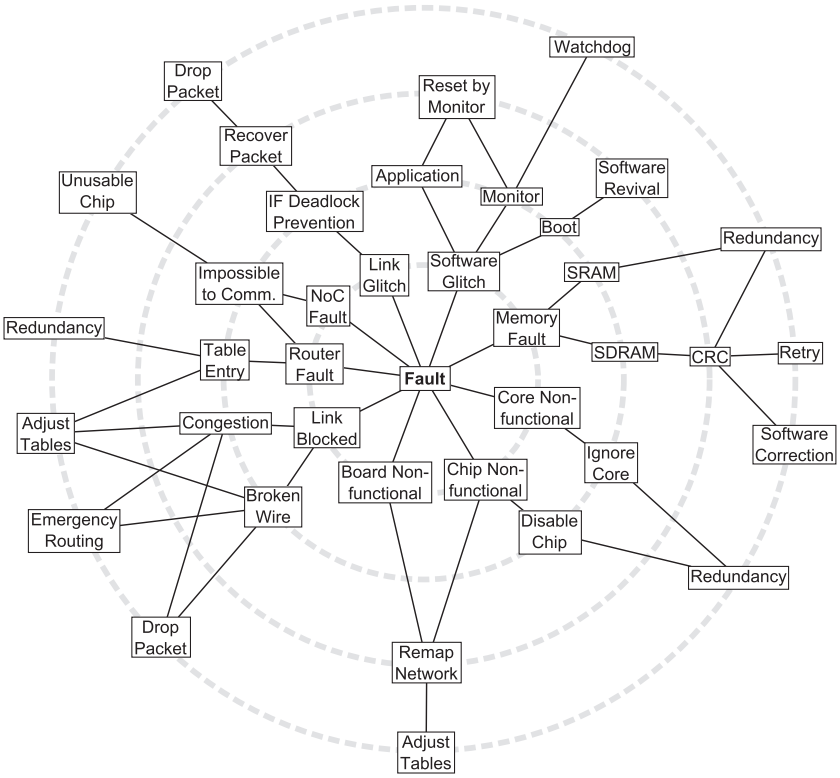



**Fig. 5.** Map of fault-tolerant features.

than its anticipated store requirement, thus there is capacity to test and map around any dubious region. This is one of the tasks for the local Monitor Processor.

In addition to redundancy, a number of features have been included either as design considerations or specifically for fault-tolerance. Fig. 5 shows a map of the expected failure types and the mechanisms provided to reduce their impact in the usability of SpiNNaker.

## 5. Diagnostics and dynamic configuration

System routines can clearly be split into two: (i) power-on testing and initial configuration, and (ii) isolation and reconfiguration during normal operation. In either case, the interaction between hardware and system software in each chip is coordinated by the Monitor Processor which maintains a continuously updated state (good, fault, disabled, *etc.*) of the chip components. The System Controller can disable or reconfigure chip components. In extreme failure cases the System Controller can be accessed from a neighbouring SpiNNaker chip using a local debug facility.

### 5.1. Power-on diagnostics and configuration

Each SpiNNaker chip performs diagnostics and initialization using minimal system software stored in the Boot ROM. In this stage each processing core performs a power-on self-test and initialisation of its private peripherals. Healthy cores then compete to access the System Controller monitor election register, the winner becoming the Monitor Processor. The remaining cores simply register their state in the System Controller and stall until the Monitor completes the node configuration (including detailed chip-level tests, initialising shared resources and detecting any connected Ethernet port). All chip-level results are stored in the System Controller.

After this step, nodes enter a listening mode awaiting external instructions. The host machine designates one or more Ethernet attached nodes to receive the system image to be executed by the Monitor Processors. The image is transmitted in blocks to the Ethernet attached Monitors which compile the image, perform a CRC check and copy it to their local memory where it can be executed. The system image informs the host machine and propagates itself to its neighbours; these neighbours send it forward their neighbours, and so on. This way the system image is flood-filled in a redundant manner as each chip will receive several copies of the system image (see below). Once system boot is complete, the Monitor Processors test connections to neighbouring chips to record any faulty link or neighbour.

The host nominates one Ethernet-attached chip as the *Reference Chip*, making it the origin address, $(0,0)$, of the network, notifying it of the topological characteristics, such as the number of chips. The Reference Chip then broadcasts its address to its six neighbours, and so on. This generates a second wave through the network that enables each chip to compute its relative address in the network topology and configure point-to-point routing tables.

#### 5.1.1. Evaluation of flood filling policies

Data loading can be done via several *flood-fill* strategies, each offering different performance and fault resilience compromises. Several of these strategies were evaluated previously [17], but when that evaluation was performed, *broadcast* packets were addressed to all neighbours and consequently most strategies had to use point-to-point (*unicast*) packets, with the consequent overloading of the injection ports. To overcome this overhead a selective multicast able to forward packets to a subset of the neighbours was included in the final design.

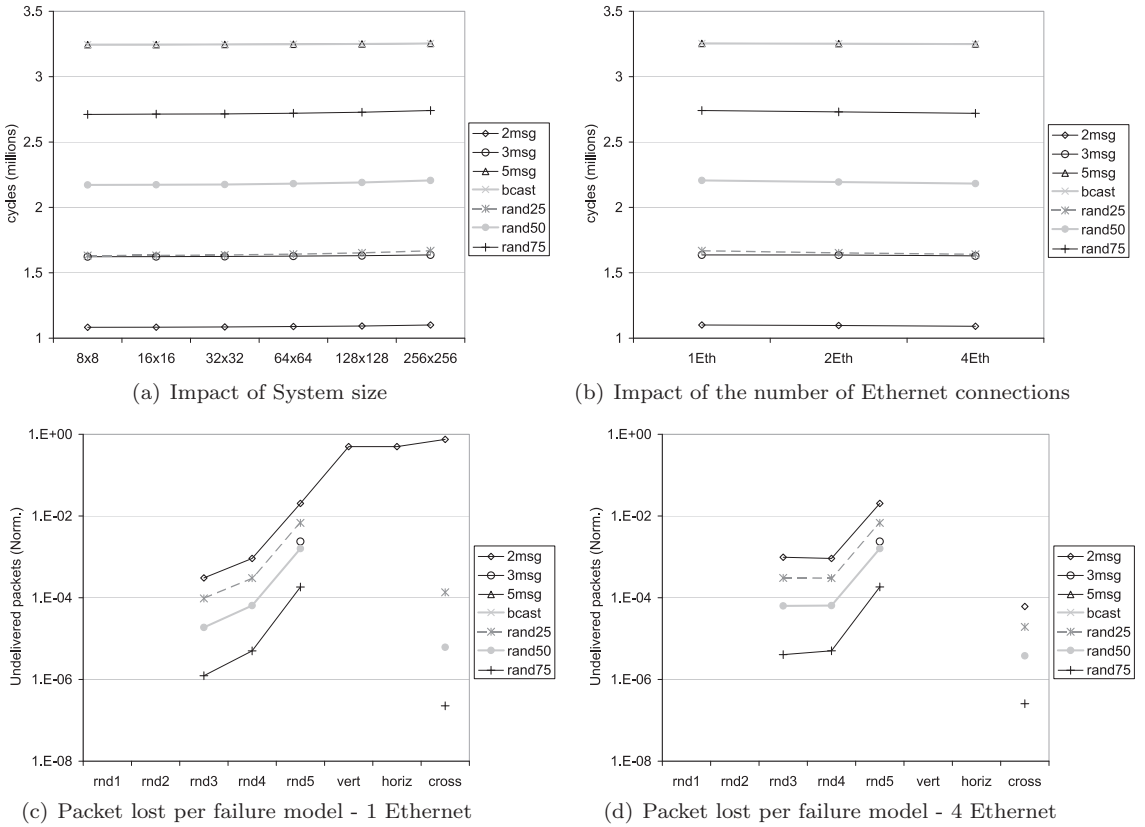
The following evaluation considers several strategies using this selective multicast. Fig. 6 summarizes the results of an event-driven simulation of the application loading process in the largest system configuration ($256 \times 256$ nodes). The top two graphs consider SpiNNaker systems without failures and are intended to show the performance (time consumed in the floodfill). The bottom two consider systems with different link failure configurations and show the resilience level provided by each strategy. Next we explain how these graphs can be interpreted.

Seven different flood-fill policies were considered in the simulations:

- *bcast* sends the packet to all neighbouring chips.
- *2msg* sends the packet only to the neighbours in the positive *X* and *Y* directions. This is the minimum number of neighbours required to perform an efficient flooding.
- *3msg* sends the packet to the neighbours in positive *X*, *Y* and *XY* diagonal.
- *5msg* sends the packet to all the neighbours but the one the original packet was received from.
- *randP* sends the packet in the positive *X* and *Y* directions and in addition randomly to each of the other directions with a P% probability. We considered 25% (*rand25*), 50% (*rand50*) and 75% (*rand75*) in our evaluation.

In the simulations considering several Ethernet ports, nodes located at (0, 0), (128, 128), (128, 0) and (0, 128) are connected to the host.

The results without failures in Fig. 6(a) and (b) show that (i) different flooding strategies provide diverse performance levels, (ii) given the 2D-pipelined nature of the application loading procedure, the loading times are not affected substan-




(a) Impact of System size

(b) Impact of the number of Ethernet connections

(c) Packet lost per failure model - 1 Ethernet

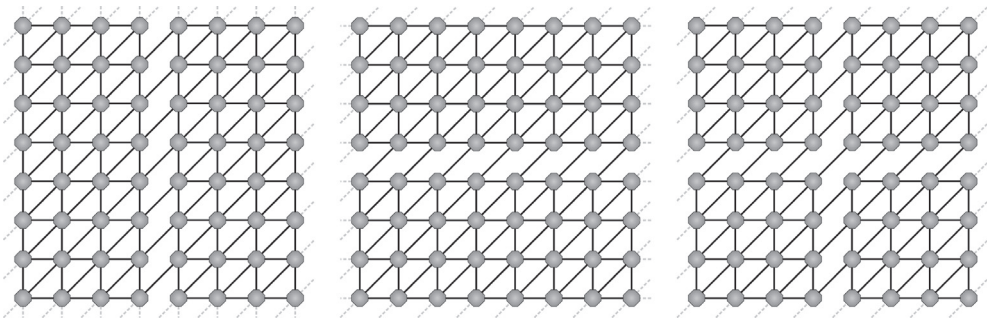
(d) Packet lost per failure model - 4 Ethernet

**Fig. 6.** Results of simulating the application loading process.

tially by the network size; and (iii) similarly, the number of Ethernet-connected nodes does not affect significantly the time required to load the application.

The configurations with failures – see Fig. 6(c) and (d) – present the normalized number of undelivered packets. Points that are not shown in the plot mean that the loading process was successful. The failure distributions considered in this study are the following.

- *vert* represents a configuration where all the links along the *Y*-axis in the bisection are treated as faulty, leading to a network split in vertical columns.
- *horiz* represents a similar configuration, but affecting the horizontal axis.
- *cross* represents the union of *horiz* and *vert*. Small-scale examples of these three configurations are shown in Fig. 7.
- The remaining configurations represent uniform random sets of link failures: 1536 (*rnd1*), 3072 (*rnd2*), 6144 (*rnd3*), 12,288 (*rnd4*) and 24,576 (*rnd5*).



(a) *vert* (b) *horiz* (c) *cross*

**Fig. 7.** Examples of the cuts used in this experimental work. Diagonal links are not cut, so the network is not completely split.

In general, those strategies sending more packets are less likely to lose packets but at the price of increasing the time required to finalize the whole process. In all cases, it will take only from 5 to 15 ms to load an application completely. Results also show that increasing the number of Ethernet connections improves robustness, especially in scenarios with multiple failures.

We conclude that the SpiNNaker configuration process is efficient, scalable and robust. Moreover, there is a reasonable range of distribution strategies that allow trade-offs between speed and fault-resilience.

### 5.2. Run-time reconfiguration

During regular operation, the Monitor Processor periodically checks and updates the state of chip resources, including the state of the links to its neighbours, in the System Controller. Any Monitor Processor can activate the *neighbour diagnostic and recovery* routine if it suspects a neighbour chip is not working properly. This *nurse* Chip will 'peek and poke' the remote System Controller to identify any healthy cores. It will first try to change the remote Monitor Processor, then try to overcome a Boot ROM failure by copying the boot-up code to the remote System RAM and remapping the remote Boot ROM and System RAM. Finally, the Nurse Chip will reset the remote chip to attempt to recover from a transient fault. If nothing works, the failed chip is isolated by disabling its clocks.

When cores or chips are detected to be faulty, the system tries to migrate their functionality (typically neurons) to other cores. This process is in principle straightforward, however depending on the failure some of the neural information may be impossible to recover. For example, to migrate from a chip because it cannot access its SDRAM, only the neural information stored in local memory can be recovered. The way to regenerate *unrecoverable* information will depend on the executed application but, as discussed before, losing neurons is acceptable in a biological brain and therefore it may be so in the simulated application.

When routers or links stop working properly, part of the routing tables may need to be reconstructed dynamically to avoid unreliable areas of the network. When a route is destroyed, the system can generate new routes by the back propagation of a routing key from the destination node to the source node. The host system will collect the required information from this procedure and will generate and propagate the updated routing tables. Alternatively, a distributed reconfiguration may rely on the Monitor Processes around the failing components for the generation and propagation of the updated routing tables.

Finally, each SpiNNaker chip is provided with a watchdog timer which detects when the Monitor Processor has not responded for a long time. When this is detected, the recovery process first tries to recover the Monitor Processor by soft resetting it. If this measure does not solve the problem, then it hard resets the chip, forcing the System Controller to select another core as a Monitor Processor.

## 6. Communications fault tolerance

Given that the supported application is communication-intensive the interconnection fabric of SpiNNaker is another critical component and therefore great effort has been devoted to design a robust and stable infrastructure.

### 6.1. On-chip and off-chip communication

The *Comms NoC* connects the processing cores via a custom on-chip router offering a bandwidth of up to 1 GByte/s. The *Communications Controller* within each processing core handles packets on behalf of its simulated neurons and interfaces with the Comms NoC. Together the on-chip router and the self-timed fabric seamlessly extend on-chip communications onto inter-chip connections.

The Comms NoC has 18 ports for internal use of the processing cores and six ports to communicate with six adjacent chips (Fig. 4). External ports contain two independent, unidirectional self-timed chip-to-chip interfaces, one for transmitting and the other one for receiving data; i.e. a failure in a link or interface only affects one of the directions. Asynchronously arriving packets to the router are arbitrated and serialised. The router can process one packet per clock cycle. It is expected that the *average* traffic demand will be much lower than this. In the event of a 'collision' packets can be delayed arbitrarily and buffering between routers helps to accommodate this. Packets are checked for integrity on arrival at the router; faulty packets are dropped into a register where they can be examined by the local Monitor Processor. Faults may be caused by corruption in transit – indicated by parity and framing errors – or by being outdated.

AER packet routing is done with a 1024 entry associative look-up table. Each table entry has its own bitmap mask that will be applied to the source address before it is compared with the table entries in order. If an address is not found then the packet is *default* routed to the port opposite the one it came from. Table entries are therefore only used when packets turn or bifurcate. Each table entry can be deactivated independently if not functioning properly; there is therefore some flexibility (potential redundancy) in the way table entries are used. As the table uses standard cell latches the soft error rate is expected to be very low.

Packets may be replicated to any subset of the router's outputs. They are sent when *all* the desired outputs are ready to accept them, stalling until this time. In the event of an output being blocked this could cause problems and backlog the rou-

ter. Thus, after a programmable interval, the router attempts to route around any blocked (or broken) external links through a so-called *emergency route*. If this still fails after another programmable interval the offending packet is dropped into a local register and the subsequent packet is tried instead. The Monitor Processor may be interrupted to examine the dropped packet and resend it – perhaps suitably modified – later.

Emergency routed packets take advantage of the triangular topology to try to reach their destinations, as shown in Fig. 8. Emergency routes are always adjacent to the intended path and the subsequent turns are therefore predefined. This is coded into the short packet header.

A particular concern is the possible occurrence of (network-level) deadlock. Conventional HPC networks avoid the formation of such chains by means of complex combinations of topology, routing algorithms and flow-control techniques [18]. Given that routing is application specific a different approach to deadlock avoidance was required. As neural applications do not require delivery guarantees, a time-out based, packet-dropping mechanism suffices, provided that the proportion of lost packets is low.

### 6.2. Topological robustness

To assess the robustness of the two-dimensional triangular torus topology we tested how it loses connectivity in the presence of link failures. A typical manufacturing process can be expected to produce components with a functional life of well over 10 years. With a very pessimistic scenario model of a 5-year mean time to failure (MTTF) with sigma of 2 years, the expected number of link failures in a complete SpiNNaker system (65,536 nodes) for any given day ($F_{\text{day}}$) would lie between 160 and 360.

We contrasted SpiNNaker topology with regular two- and three-dimensional tori for 65,536 nodes. The 2-D topologies are arranged as square networks of $256 \times 256$ nodes whereas the 3-D torus is arranged as a $64 \times 32 \times 32$ network.

We assessed how the three topologies lose node-connectivity as the number of link failures increases from 1 to 65,536. A depth-first search algorithm was used to calculate this figure for $10^5$ random uniform failure configurations. The average of these $10^5$ runs is plotted in Fig. 9. In the figure we can see that the triangular two-dimensional torus implemented in SpiNNaker provides a robustness level similar to a three-dimensional torus. Both topologies can support up to 8,192 random link failures without any of the nodes losing connectivity with the rest of the system, more than one order of magnitude above ($F_{\text{day}}$). On average tens of thousands of link failures are required to lose one or more nodes. This robustness motivates the use of the triangular torus topology in SpiNNaker.

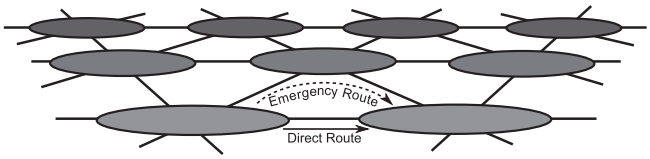



**Fig. 8.** Regular (direct) and emergency routes from a chip to a neighbour.

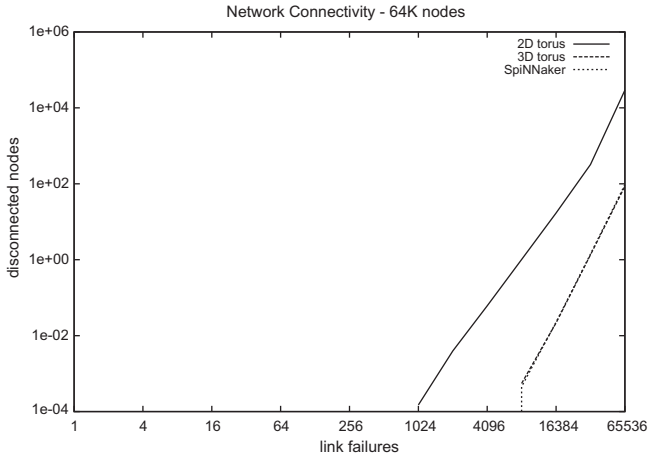



**Fig. 9.** Number of disconnected nodes in the presence of link failures of $2^{16}$-node topologies.

### 6.3. Interconnect stability under severe degradation

The packet dropping mechanism provides a deadlock-free interconnection network. However, there is a loss of information that has to be assessed. Spiking neuron systems can work when a few messages are lost but, even in very pessimistic scenarios, the number of dropped packets should be low (below 1 packet per million. Approximatelly 1 packet each 1,500 cycles in the following experiments). Simulation has verified that the network can deal with loads well-above the expected without significantly impacting applications (packet delay is acceptable) [21].

The simulations model a $256 \times 256$ network considering scenarios in which the network suffers different levels of hard failures. To account for the real-time constraint, this section investigates the temporal evolution of the system and focuses on *stability*, understood as the variability (which should be low) of the figures of merit and assesses the effectiveness of the emergency routing mechanism.

#### 6.3.1. Simulation model of the SpiNNaker network

A simplified model of the SpiNNaker interconnection infrastructure has been implemented in INSEE [19]. It includes the topological description of the system and a model of the router.

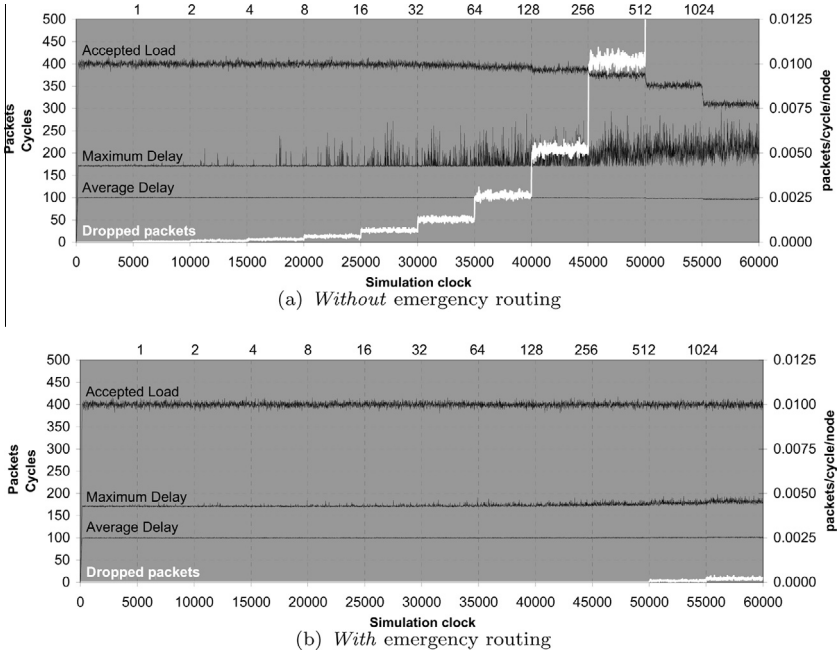
Time is modelled in terms of abstract network *cycles* the time to route and forward a packet (1 network cycle $\approx$ 10 processor cycles). A network node represents a complete SpiNNaker chip, with all its cores and its router. Nodes are modelled as independent traffic sources that inject packets following a Bernoulli temporal distribution that can be parameterized to generate any chosen injection rate. The spatial distribution of the traffic is uniform.

All ports are modelled as a single four-packet queue. If this is full and the node tries to inject a packet, it is dropped. Communications are point-to-point. Routing tables are not implemented, using Dimension Order Routing instead. This emulates the expected shape of communications – two straight lines with one inflection point [20]. As discussed in Section 5, the SpiNNaker system is aware of network failures and can modify its routing tables to avoid conflictive areas. In contrast, DOR is oblivious and therefore unaware of network failures so our results should be taken as pessimistic

The experiments consider systems with 0 to 1024 link failures which covers scenarios well above $F_{\text{day}}$ (as discussed before). Consequently this evaluation should be understood as a *worst-case* study.

#### 6.3.2. Experiments and discussion of results

In the following experimental work, we will use the maximum network load expected during regular operational levels of SpiNNaker which was derived in previous research [21]. We show the evolution of a SpiNNaker network degrading progressively from 0 to 1024 random link failures which are introduced at the beginning of every sampling period (5,000 cycles). The figures of interest are accepted load, number of dropped packets and packet latency figures (average and maximum). Router parameters are fixed to the values suggested by previous experiments [21]. To assess the impact of emergency routing on system stability, we plot results *without* (Fig. 10a) and *with* this mechanism (Fig. 10b).




(a) *Without* emergency routing

(b) *With* emergency routing

**Fig. 10.** Temporal evolution of accepted load, latency figures and dropped packets for a $256 \times 256$ gradually degrading SpiNNaker system.

To better understand the graphs, notice that the *X*-axis measures time (cycles). The labels at the top (1, 2, 4, ..., 1024) indicate the total number of failures at the corresponding time: during the first 5 kcycles the network is fully operative, from 5k to 10k there is a single link failure, from 10k to 15k there are two failures, and so on. Each performance metric has its own unit, indicated in the *Y*-axes: packets (for the dropped packets line), cycles (for the average and maximum delay lines) and packets/cycle/node (for the accepted load line). Note that plotted data, including the number of dropped packets, are not cumulative, but correspond to 10-cycle measurement periods.

Fig. 10(a) shows how the progressive introduction of failures results in a high variability of performance metrics when emergency routing is not activated. Accepted load drops by up to 25%, maximum delay noticeably fluctuates and the number of dropped packets grows linearly with the number of link failures. For clarity, in the graphs the *Y*-axis is bounded by 0 and 500, which leaves out the number of dropped packets for 512 failures (around 800) and 1024 failures (around 1600). We can see that even with a single failure the system with emergency routing deactivated (approx. 1 packet dropped every 50 cycles) noticeably exceeds the acceptable limit.

In contrast, evolution with the emergency routing activated, Fig. 10(b), shows very stable performance metrics. Only some minor peaks in the maximum delay can be observed. The most remarkable difference is in the number of dropped packets: no packets are dropped for experiments with fewer than 512 failures. Considering that these scenarios are well beyond the described pessimistic range of failures (160–360), we can confirm that the emergency routing plays a major role in improving fault tolerance at the network level. It is also worth noticing that, in all cases, when failures are introduced in the system we do not observe significant transient periods. This means that, after a failure, the system reaches a stable situation very rapidly.

The conclusion is that the SpiNNaker interconnection network provides a highly stable communications fabric for the real-time simulation of spiking neurons. Even under very pessimistic scenarios the interconnection network does not show significant performance fluctuations and degrades gracefully.

### 6.4. Chip-to-chip interfaces

The self-timed communication fabric is implemented using handshake protocols because of their advantages for large networks:

- Chips can be interconnected without regard to wiring delays which simplifies machine construction as some chips will be adjacent on a PCB whilst others may require considerable cabling or buffering with potential for delays and skew.
- Power economy by limiting logic transitions (no clock information is transmitted).
- Adequately high speed is retained.
- Well suited to short, intermittent transmissions – appropriate for neural communications.

There is, however, a significant drawback to handshaking links: in the presence of noise they are prone to deadlock. In this subsection these are deadlocks at the *interface* level, not to be confused with the previously discussed network-level deadlocks.

A *handshake link* can be thought of as passing a data token from the sender to the receiver which the handshake returns so that the next data can be sent. Noise on the link may not only corrupt the data but also this control information, removing or introducing other tokens so that the sender and receiver lose coherence. In the most serious case, a lost token can result in each waiting for the other and the data link cannot recover. Timeout is not possible as there is no concept of time, only sequencing.

The on-chip network uses Silistix CHAIN [23] interconnection with 3-of-6 return-to-zero (RTZ) coding [22]. This provides a convenient symbol set with 20 codes of which 16 are used. A separate channel provides an End-of-Packet (EoP) marker. The inter-chip links use a different protocol to balance speed, pin usage and (particularly) power consumption. Each four bit token is encoded as a 2-of-7 code (21 possible codes of which 17 are used: 4 bits plus EoP [24]). To reduce the number of transitions a non-return-to-zero (NRZ) coding is used.

Noise glitches on the inter-chip wires introduce extra transitions, potentially in both the forward and return paths; these must be detected and recovered from at each end of the link. The off-chip wiring is the most likely place for noise to be induced and it is assumed that such noise will cause a short glitch (*i.e. two* extra transitions) on a wire. Glitches will be reasonably uncommon, therefore data integrity is not addressed in hardware; detection of damaged packets can be delegated to system software if any recovery is to be attempted. The hardware simply has to keep running.

The majority of the fault tolerance resides in the receiver (Fig. 11) where various stages filter out potential problems.

- *NRZ to RTZ Conversion*: The problem for the first stage of the receiver is that it may not know the *level* of an input wire at its reset time. This is overcome using a phase converter comprising two parallel RS flip-flops (Fig. 12) which acts as a transition detector which is set by one *or more* input transitions. It is cleared locally between the detection of a symbol and its external acknowledgement. Two transitions are made per symbol. When *at least* two such phase converters are set, it is assumed that an input flit has been captured and passed to the next stage.

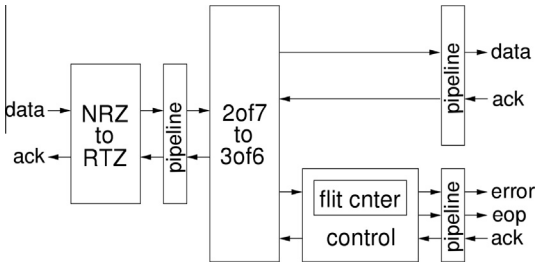



**Fig. 11.** Inter-Chip link receiver.

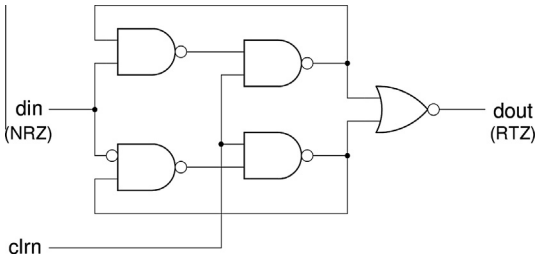



**Fig. 12.** Phase-Insensitive NRZ-RTZ converter.

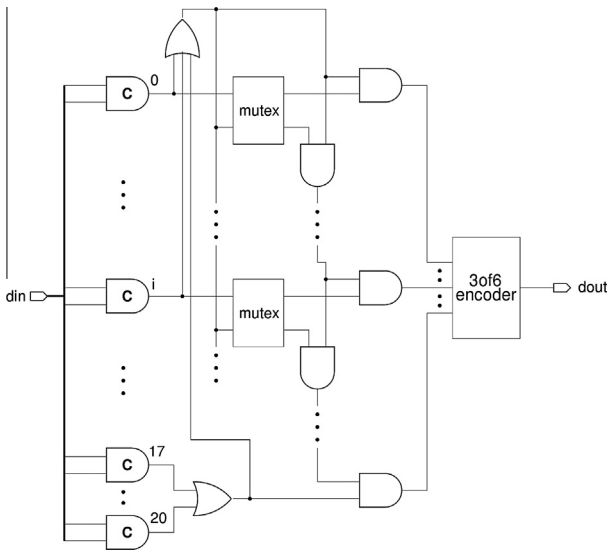



**Fig. 13.** Fault-tolerant symbol converter.

- *2-of-7 to 3-of-6 Conversion*: The flit is then searched by the symbol converter (Fig. 13) using an asynchronous state machine with Muller C-elements [25,26]. In the absence of errors exactly one of these will be set but a glitch may have set more. The output is therefore filtered with a priority encoder based on mutual exclusion elements which chooses a single, legal, 'one-hot' code. There is no attempt to choose the 'correct' code – that information is not available – but *any* legal code will prevent a deadlock. It is then a simple matter to generate a 3-of-6 code with an auxiliary EoP line appropriate for the NoC.
- *Flit Counter*: Glitches can easily insert extra flits into a packet but it is important that no packet exceeds a maximum length. A flit counter is added to keep track of the number of flits and, if it exceeds a given threshold an extra EoP is inserted, notifying explicitly a framing error. The counter is reset on reception of an EoP.
- *Transmitter*: The only external input is the handshake acknowledge line. A phase converter detects at least one transition and treats that as the acknowledgement, further transitions being ignored until sending the next flit. When a transmitter is reset its state is 'ready to send'. Similarly, when a receiver is reset it sends an acknowledgement. If the reset occurred

**Table 2**
Simulation results for chip-to-chip interface designs.

| Protocol converter | Conventional (%) | Fault-tolerant (%) |
|---|---|---|
| Glitches | 47.98 | 48.54 |
| Error-free packets | 91.97 | 82.96 |
| Deadlocks | 1.60 | 0.00 |

during reception this enables the transmitter to send again but if no acknowledgement was outstanding then there is a spurious transition which is filtered by the transmitter. The glitch tolerance automatically provides a simple, robust, single-ended reset capability which means that a node may be reset independently and still recover communication with its neighbours.

*6.4.1. Performance assessment*

Our novel fault tolerant interface was compared with a conventional unit. Both circuits were simulated in Verilog handling roughly a million packets each in an extremely noisy environment in which packets have a 50% probability of being affected by a glitch. This noise level is exceedingly high and thus the number of packets corrupted should not be a concern. Results of these simulations are shown in Table 2. *Glitches* represents the actual packet ratio affected by a glitch. *Error-free Packets* represents the percentage of the packets affected by a glitch that were interpreted correctly i.e., those that have resisted the glitch; *Deadlocks* represents the percentage of packets affected by a glitch that deadlocked the interface.

As expected our design did not deadlock whereas a conventional unit deadlocked roughly 2% of the times that a glitch appears. This is very significant as a single deadlock has the potential to cripple a link *permanently* (until the whole system is rebooted). Given the communication-intensive application model supported by SpiNNaker this would mean a network becoming highly degraded very quickly if glitches appeared.

The price of the deadlock-free interface is that glitches alter the received data roughly 10% more often. This is acceptable as glitches should be rare and erroneous packets can be detected and dropped.

### *6.5. Intra-chip connections*

The asynchronous on-chip links are much less sensitive to noise-induced glitches, so they employ simpler logic. However the potential for deadlock with handshake communications still applies. To alleviate this intra-chip interfaces are provided with two levels of reset (soft and hard). Watchdog will apply soft reset first and if this does not solve the problem, it will perform a hard reset of the entire node, thus disrupting chip operation – but not deadlocking the on-chip network.

## 7. Other fault-tolerant features

### *7.1. Clock redundancy*

SpiNNaker chips have two independent PLLs with the intention of running the processors and the router at one frequency (200 MHz) and the SDRAM at another (166 MHz) to improve overall performance. However clock sources can be switched so, in the event of a PLL failure, all subsystems' clocks can be derived from the same source. This may reduce performance on that node but has no other consequence as the GALS interconnection is inherently adaptive.

### *7.2. GALS implications on fault tolerance*

Using a GALS approach not only facilitates the SoC design process but also simplifies isolation of faulty components in run time by supporting resetting or disabling on-chip components independently from the rest of the SoC.

### *7.3. Memory subsystem fault tolerance*

Various RAM is spread across a SpiNNaker chip (private TCMs, on-chip SRAM and off-chip RAM) which, as explained before, are the main expected points of failure.

In principle it is possible to work-around hard failures in private memories but the degree of redundancy in the system means our plan is simply to inactivate cores with permanent TCM failure(s). The SDRAMs have some spare capacity and are mapped by the Monitor Processor so hard failures can be worked around.

Soft errors present a different challenge. The SDRAM is usually accessed in blocks via DMA. The DMA controller includes a fully programmable CRC generator and checker so faulty blocks can be detected and subject to software recovery.

Private memories are not protected by error detection; this was a pragmatic decision to maximise the speed and capacity of the chip. Soft errors in code space can cause a software crash resulting in some lost neural information. Recovery from this should be achieved by the watchdog mechanism resetting the affected core. Corruption of data space may be detected by

software limit checks or crashing or other erratic behaviour. There may be some loss of data or some erroneous neural firing but the application should be robust enough to withstand this; it almost certainly happens in biological systems too!

### 7.4. Connecting with the outside world

A SpiNNaker system communicates with a *host computer* via Ethernet (Fig. 1) using TCP/IP. This communication is used for different management actions, such as loading to the chip memories the application code or the neural connectivity information. Although all chips have an Ethernet interface in practice only a few will make use of it to reduce power consumption and maximize the computing resources available to neurons. Each Ethernet-connected chip translates frame-based communication to inter-chip packet-based communication. The presence of several possible interfaces does, however, eliminate another possible single-point failure.

## 8. Related work

Research in simulating biologically-plausible neural networks (brain-like systems) is not new. In the early 1990s a team at U.C. Berkeley worked on the Connectionist Network Supercomputer [27] which aimed to build a supercomputer specifically tailored for neural computation as a tool for connectionist research. The system was a 2D mesh, with a target size of 128 nodes (scalable to 512). Each node would incorporate a general-purpose RISC processor plus a vector coprocessor, 16 MBytes of RAM and a router. As far as we know, the node was built (under the codename T0), but the system never operated as a network. Experiments using up to five nodes in a bus configuration were discussed in [28].

More recently, the Microelectronics Division at the Technical University of Berlin worked on a project entitled *Design and implementation of spiking neural networks* [http://mikro.ee.tu-berlin.de/spinn] whose objectives are similar to those of SpiNNaker. A product of this is the Spiking Neural Network Emulation Engine (SEE), an acceleration board implemented with FPGAs interconnected via an on-board bus. SEE accelerators were able to perform neural computations 30 times faster than a desktop PC [29]. However, as these boards cannot be connected to form a network, they are not able to scale to the magnitudes of SpiNNaker.

Research on spiking neural networks has also used different *off-the-shelf* technologies such as FPGAs [30], graphic processors [31] and general purpose processors and accelerators [32], obtaining speed-ups of over two orders of magnitude compared to software-only implementations.

The relatively small scale of these systems allowed the assumption of a complete absence of component failures and, therefore, did not address reliability issues and did not incorporate fault-tolerant techniques.

As far as we know, there are only three active projects comparable to SpiNNaker in terms of simulation scale. First, the Blue Brain project [http://bluebrain.epfl.ch/] aims to create *biologically accurate* functional models of the brain; however, model complexity (far more intricate than SpiNNaker's) only allows real-time execution of roughly a neuron per node [33]. This is a low figure in comparison with the several thousand (simpler) neurons per node supported by SpiNNaker.

Secondly, DARPA's System of Neuromorphic Adaptive Plastic Scalable Electronics (SyNAPSE) project claims that it has achieved the simulation of spiking neural networks the size of a cat's brain [34] – $10^9$ neurons – using Izhikevich models like those supported by SpiNNaker. However their simulations run 2-3 orders of magnitude slower than real-time.

In contrast with the biologically-inspired SpiNNaker architecture, neither Blue Brain nor SyNAPSE contemplate the construction of a custom architecture but use general-purpose supercomputers from the IBM BlueGene family, depending on the underlying platform for their reliability and fault tolerance.

The IBM BlueGene/L consists of 64 K compute nodes, each based on PowerPC 400 processors. Additionally, it contains several service nodes that reside outside the core [35] and communicate with it using Ethernet. This infrastructure is used for booting, controlling and monitoring the system. System monitoring and job execution is done by the combined action of service and I/O nodes which maintain log files. During boot-up service nodes can control the computing core to the lowest level of granularity [36]. Service nodes can also directly write to and read from the device control registers of each processor. This feature is useful for handling runtime problems and investigating any booting up issues. For fault tolerance at boot-up or at run-time a self-test mechanism is kept in each chip to perform system diagnostics. The BlueGene supercomputer or others, *e.g.*, the Cray XT family of supercomputers [37,38], being general-purpose will provide solutions that do not match the power-efficiency of SpiNNaker.

Last but not least, the FACETS project [39] is attempting to create a faster than real-time hardware system for the simulation of networks of large but unspecified size. This architecture, while biologically inspired, uses a fixed synapse and neuron model and, therefore, is not a system as general as SpiNNaker. It employs *analogue* circuits to implement most of the central dynamic functions. For these blocks what would constitute a 'fault' is not precisely defined since analogue circuits exhibit a continuum of states. It is therefore relevant to discuss fault tolerance only with respect to the digital components: the communications infrastructure of the design. The FACETS architecture uses wafer-scale devices [40] to achieve the necessary connectivity. It uses AER signalling (similar to SpiNNaker), but with a circuit-switched, synchronous communications subsystem. Systems of this kind are fault tolerant in the sense of being reconfigurable in the event of a failed link; however they are not live-reroutable, thus the system provides no protection against transient faults nor does it permit packet recovery or retransmission while the system is active. A failed link requires at least a local reconfiguration with possible further

routing impact. FACETS authors discuss fault tolerance but only as a general property of neural systems; the system does not include specifically designed fault-tolerant mechanisms. Thus the FACETS system, and its associated HICANN devices, once again represent a very different system designed to solve a different problem: faster than real-time neural simulation, for which power consumption is not a factor and fault tolerance merely a side effect rather than a design feature.

Outside of the field of brain-like systems we can cite a heterogeneous SoC with certain architectural similarities to SpiNNaker [41]. This consists of an array of processors connected over an on-chip NoC and containing various heterogeneous system components. However, that project considers general-purpose applications within mission critical scenarios requiring the robustness of triple modular redundancy. This approach is an expensive solution unsuitable for SpiNNaker. In addition, their NoC appears to be a conventional synchronous design rather than the SpiNNaker self-timed communication fabric which may difficult scaling up the system.

Reviewing the literature on general purpose multiprocessor systems we can see how memory fault tolerance efforts have been devoted mainly to the interconnect structure [42], and to the use of ECC (originally following [43]), although this may not be in itself sufficient [44]. Given that symmetric redundancy of memory is expensive, recent work has introduced the concept of heterogeneous fault tolerance: graceful fall-back onto other components able to perform the same function, possibly with reduced performance [45,46]. Such an approach lowers overall hardware costs and represents a reasonable compromise in a power- or area-constrained design. Our asymmetric memory architecture follows this approach.

Implementing fault tolerance in direct networks (such as 3D tori) is complex and costly and, therefore, a hot research topic. Current solutions are neither easy nor cheap to implement in silicon (see, for example, [47,48]). The simple emergency routing mechanism implemented in SpiNNker has been shown to be very effective for this purpose.

## 9. Summary and conclusions

This paper has focused on introducing the broad collection of fault tolerance mechanisms implemented in SpiNNaker. Such features are quite extensive, and we have presented descriptions of the principal mechanisms and, where available, the pre-silicon assessment of their effectiveness. Some of the most important features discussed in this paper are the following:

- A collection of system routines able to detect faults and to quickly recover from them when possible or to isolate components and to reconfigure the system otherwise.
- A range of application loading policies offering different levels of resilience and performance which can be used depending on the level of system degradation.
- The use of GALS logic that facilitates the SoC design process, simplifies timing closure and simplifies the isolation of faulty components but introduces weaknesses that have been overcome with custom-hardware.
- Asymmetric redundancy of the memory subsystem, granting graceful fall-back onto other components able to perform the same function although with reduced performance. Such an approach lowers overall hardware costs and represents a reasonable compromise in a power- or area-constrained design.
- A novel robust self-timed chip-to-chip interface circuit, resilient to noise-induced glitches preventing deadlocks.
- A stable communication fabric able to support communication demands exceeding those expected during regular operation.
- The novel emergency routing mechanism helps to deal with congestion and network failures.

The main conclusion of this paper is that SpiNNaker is a well-balanced fault-resilient architecture in which fault-tolerance has been considered a fundamental foundation of its design. This should facilitate its scaling from the prototype, 4-chip systems into practical, large-scale networks with over 1 million cores.

## Acknowledgements

The SpiNNaker project is supported by the Engineering and Physical Sciences Research Council (EPSRC), through grants EP/G015740/1, EP/G013500/1, EP/D07908X/1 and GR/S61270/01, and also by industrial partners ARM and Silistix. Prof. Miguel-Alonso is supported by the Spanish Ministry of Science and Innovation (grant TIN2010-14931) and by the Basque Government (grant IT-242-07). Dr. Luján holds a Royal Society University Research Fellowship. Dr. Navaridas was a Royal Society Newton International Fellow when this research was performed.

## References

[1] S. Davies, C. Patterson, F. Galluppi, A.D. Rast, D. Lester, S.B. Furber, Interfacing real-time spiking I/O with the SpiNNaker neuromimetic architecture, in: Proceedings 17th International Conference (ICONIP 2010), 2010.
[2] T. Elliott, N. Shadbolt, Developmental robotics: Manifesto and application, Philos. Trans. Royal Soc. A (361) (2003) 2187–2206.
[3] S. Herculano-Houzel, The human brain in numbers: a linearly scaled-up primate brain, Front Hum Neurosci. 3 (0). doi:10.3389/neuro.09.031.2009.
[4] M. Nicolaidis, Soft Errors in Modern Electronic Systems, first ed., Springer, 2011.
[5] C. Constantinescu, Trends and challenges in VLSI circuit reliability, IEEE Micro 23 (4) (2003) 14–19, http://dx.doi.org/10.1109/MM.2003.1225959.
[6] D.M. Chapiro, Globally-Asynchronous Locally-Synchronous Systems, Stanford University, CA, Ph.D. thesis, 1984.

[7] E. Izhikevich, Simple model of spiking neurons, IEEE Trans. Neural Networks 14 (2003) 1569–1572.
[8] C. Koch, I. Segev, Methods in Neuronal Modeling, The MIT Press, 1989.
[9] A.D. Rast, J. Navaridas, X. Jin, F. Galluppi, L.A. Plana, J. Miguel-Alonso, C. Patterson, M. Lujan, S. Furber, Managing burstiness and scalability in event-driven models on the SpiNNaker neuromimetic system, Int. J. Parallel Program 40 (6) (2012) 553–582.
[10] F. Rosenblatt, Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms, Spartan Books, Washington, 1962.
[11] P. Dayan, L.F. Abbott, Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems, The MIT Press, 2005.
[12] P. Tiesinga, T. Sejnowski, Precision of pulse-coupled networks of integrate-and-fire neurons, Network 12 (2) (2001) 215–233.
[13] H. Brody, Cell counts in cerebral cortex and brainstem, Alzheimer Disease Senile Dementia and Related Disorders (1978) 345–351.
[14] W. Maass, C.M. Bishop, Pulsed Neural Networks, The MIT Press, 1998.
[15] M.A. Sivilotti, Wiring Considerations in Analog VLSI Systems, with Application to Field-Programmable Networks (VLSI), Ph.D. thesis, California Institute of Technology, Pasadena, CA, 1991.
[16] M. Mahowald, VLSI Analogs of Neuronal Visual Processing: a Synthesis of Form and Function, Ph.D. dissertation, California Institute of Technology, Pasadena, CA, 1992.
[17] M. Khan, A. Rast, J. Navaridas, X. Jin, L. Plana, M. Lujn, S. Temple, C. Patterson, D. Richards, J. Woods, J. Miguel-Alonso, S. Furber, Event-driven configuration of a neural network CMP system over an homogeneous interconnect fabric, Parallel Computing, in press, corrected proof.
[18] W. J. Dally, B. Towles, Principles and Practices of Interconnection Networks, Morgan Kaufmann, 2004.
[19] J. Navaridas, J. Miguel-Alonso, J. Pascual, F. Ridruejo, Simulating and evaluating interconnection networks with INSEE, Simulation Modelling Practice and Theory 19 (1) (2011) 494–515.
[20] 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 2008 International Joint Conference on Neural Networks (IJCNN), 2008.
[21] J. Navaridas, M. Luján, J. Miguel-Alonso, L. Plana, S. Furber, Understanding the interconnection network of SpiNNaker, in: Proceedings of 23rd International Conference on Supercomputing (ICS), 2009, pp. 286–295.
[22] T. Verhoeff, Delay-insensitive codes an overview, Distrib. Comput. 3 (1988) 1–8, http://dx.doi.org/10.1007/BF01788562.
[23] J. Bainbridge, S. Furber, CHAIN: a delay-insensitive chip area interconnect, IEEE Micro 22 (5) (2002) 16–23.
[24] Y. Shi, S. Furber, J. Garside, L. Plana, Fault-tolerant delay insensitive inter-chip communication, in: Proceedings of 15th IEEE International Symposium on Asynchronous Circuits and Systems, 2009, pp. 77–84.
[25] D.E. Muller, W. Bartky, A theory of asynchronous circuits, in: Proceedings of International Symposium Theory of Switching, Part 1, Harvard Univ. Press, 1959, pp. 204–243.
[26] I. Sutherland, R. F. Sproull, D. Harris, Logical Effort, Morgan Kaufmann, 1999.
[27] K. Asanovic, J. Beck, J. Feldman, N. Morgan, J. Wawrzynek, A supercomputer for neural computation, in: Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN), vol. 1, 1994, pp. 5–9.
[28] P. Pfaerber, K. Asanovic, Parallel neural network training on MultiSpert, in: Proceedings of IEEE 3rd International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), 1997.
[29] H. Hellmich, M. Geike, P. Griep, P. Mahr, M. Rafanelli, H. Klar, Emulation engine for spiking neurons and adaptive synaptic weights, in: Proceedings of IEEE International Joint Conference on Neural Networks (IJCNN), 2005.
[30] M. Pearson, I. Gilhespy, K. Gurney, C. Melhuish, B. Mitchinson, M. Nibouche, A. Pipe, A real-time, FPGA based, biologically plausible neural network processor, in: Artificial Neural Networks: Formal Models and Their Applications ICANN 2005, vol. 3697 of Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 2005, pp. 755–756. doi:10.1007/11550907_161.
[31] B. Han, T. Taha, Neuromorphic models on a GPGPU cluster, in: Neural Networks (IJCNN), The, International Joint Conference on 2010 (2010) 1–8, http://dx.doi.org/10.1109/IJCNN.2010.5596803.
[32] M. Bhuiyan, V. Pallipuram, M. Smith, Acceleration of spiking neural networks in emerging multi-core and GPU architectures, in: Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW), IEEE International Symposium on 2010 (2010) 1–8, http://dx.doi.org/10.1109/IPDPSW.2010.5470899.
[33] H. Markram, The blue brain project, Nature Reviews Neuroscience 7 (2006) 153–160, http://dx.doi.org/10.1038/nrn1848.
[34] R. Ananthanarayanan, S.K. Esser, H.D. Simon, D.S. Modha, The cat is out of the bag: cortical simulations with $10^9$ neurons, $10^{13}$ synapses, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09, ACM, New York, NY, USA, 2009, pp. 63:1–63:12. doi:http://doi.acm.org/10.1145/1654059.1654124.
[35] A. Gara, M.A. Blumrich, D. Chen, G.L.-T. Chiu, P. Coteus, M.E. Giampapa, R.A. Haring, P. Heidelberger, D. Hoenicke, G.V. Kopcsay, T.A. Liebsch, M. Ohmacht, B.D. Steinmacher-Burow, T. Takken, P. Vranas, Overview of the BlueGene/L system architecture, IBM J. Res. Develop. 49 (2.3) (2005) 195–212. doi:10.1147/rd.492.0195.
[36] R. Haring, R. Bellofatto, A. Bright, P. Crumley, M. Dombrowa, S. Douskey, M. Ellavsky, B. Gopalsamy, D. Hoenicke, T. Liebsch, J. Marcella, M. Ohmacht, BlueGene/L compute chip: Control, test and bring up intrastructure, IBM Journal of Research and Development 49 (2005) 289–301.
[37] P. Worley, Comparison of Cray XT3 and XT4 Scalability, Cray Inc., May 2007.
[38] A. Bland, J. Rogers, R. Kendall, D. Kothe, G. Shipman, Jaguar: The world's most powerful computer, in: Cray User Group 2009, Cray Inc., 2009.
[39] J. Fieres, J. Schemmel, K. Meier, Realizing biological spiking neural network models in a configurable wafer-scale hardware system, in: Proceedings of 2008 International Joint Conference on Neural Networks (IJCNN), 2008, pp. 969–976.
[40] J. Schemmel, J. Fieres, K. Meier, Wafer-scale integration of analog neural networks, in: Neural Networks, 2008. IJCNN 2008. (IEEE World Congress on Computational Intelligence). IEEE International Joint Conference on, 2008, pp. 431–438. doi:10.1109/IJCNN.2008.4633828.
[41] R. Obermaisser, H. Kraut, C. Salloum, A transient-resilient system-on-a-chip architecture with support for on-chip and off-chip TMR, in: Proc, Seventh European Dependable Computing Conference (EDCC) 2008 (2008) 123–134.
[42] S. Mahmud, L.T. Samaratunga, S. Kommidi, Fault-tolerant hierarchical networks for shared memory multiprocessors and their bandwidth analysis, The Computer Journal 45 (2) (2002) 147–161.
[43] J. Yamada, T. Mano, J. Inoue, S. Nakajima, T. Matsuda, A submicron 1 Mbit dynamic RAM with a 4-bit-at-a-time built-in ECC circuit, IEEEJSSC SC-19 (5) (1984) 627–633.
[44] C. Chen, A. Somani, Fault-containment in cache memories for TMR redundant processor systems, IEEE Trans. Comput. 48 (4) (1999) 386–397.
[45] I. Hong, M. Potkonjak, R. Karri, A heterogeneous built-in self-repair approach using system-level synthesis flexibility, IEEE Trans. Reliab. 53 (1) (2004) 93–101.
[46] Y. Nakamura, K. Hiraki, Heterogeneous functional units for high speed fault-tolerant execution stage, in: Proceedings of 2007 13th Pacific Rim International Symposium on Dependable Computing, 2007, pp. 260–263.
[47] M. Gomez, N. Nordbotten, J. Flich, P. Lopez, A. Robles, J. Duato, T. Skeie, O. Lysne, A routing methodology for achieving fault tolerance in direct networks, IEEE Trans. Comput. 55 (4) (2006) 400–415.
[48] V. Puente, J. Gregorio, Immucube: Scalable fault-tolerant routing for k-ary n-cube networks, IEEE Transactions on Parallel and Distributed Systems 18 (6) (2007) 776–788.