

Fault tolerance mechanism implementations in SpiNNaker

Patrick Camilleri

Contents

1 Software fault tolerance	2
1.1 Introduction	2
1.2 Software Failure and Classification of Software Faults	2
1.2.1 What is a software failure?	2
1.2.2 Classification of software faults	3
1.3 Techniques for Fault Tolerance in Software	3
1.3.1 Design diversity	4
1.3.2 Data diversity	4
1.3.3 Environment diversity	4
1.3.4 Checkpointing and Recovery	5
2 Fault tolerance on SpiNNaker	6
2.1 SpiNNaker architecture	6
2.1.1 Overview	6
2.1.2 Hardware fault tolerance mechanisms in the SpiNNaker chip	6
3 Network congestion	11
3.1 Backpressure	11
3.2 Dumped-packet reinsertion	11
4 Process migration	13
5 CRC error correction	15
5.1 Introduction	15
5.2 Discrete logarithms (Wikipedia)	15
5.3 Objectives	16
5.4 SpiNNaker Overview	16
5.5 Memory	16
5.6 CRC unit	17
5.7 Programmable CRC	18
5.8 Efficient Programmable CRC Circuits	18
5.9 Algorithms for Computing Discrete Logarithms	18

1 Software fault tolerance

1.1 Introduction

With the explosive growth in Internet technology and the emergence of a number of new and advanced applications, assured availability of computer systems has become a critical issue. The challenge is to provide the desired availability and performance at a low cost. Outages in computer systems consist of both hardware and software failures. While hardware failures have been studied extensively and varied mechanisms have been presented to increase the system availability with regard to such failures, software failures and the corresponding reliability/availability analysis has not drawn much attention from researchers. The study of software failures has now become more important since it has been recognized that computer systems outages are more due to software faults than to hardware faults (Gray and Siewiorek, 1991).

It was long assumed that concepts of reliability and failure rate do not apply to software since software does not degrade physically as a function of time or environmental stresses. But it is not reasonable to expect a software system to be operating on the same input data, user requirements and computing environment constantly. These changes must be accommodated in most applications and so a history of fault-free behavior cannot be taken as an indication of fault-free behaviour in the future. Many studies have shown that even for applications which have relatively less complex software, many failures in computer systems are due to software bugs (Pradhan, 1996). Therefore, software reliability is one of the weakest links in system reliability.

Demands on software reliability and availability have increased tremendously due to the nature of present day applications. There are stringent requirements in terms of cumulative down time and failure free operation of software. In many cases, there are serious consequences like huge economic losses or risk to human life if the software is faulty. In spite of using the best available software development techniques, there have been many instances of spectacular system failures due to software errors like the crash of the Ariane 5 launcher in June 1996 which was attributed to specification and design errors in the software of the inertial reference system (Lions et al., 1996). Also, increasing complexity and proliferation of real-time software have led to a resultant increase in software failures.

A major limitation in developing reliable software is the test and verification process. It is almost impossible to fully test and verify if a software is bug-free. Formal verification techniques such as proof of correctness cannot be applied to large programs. Furthermore, the shortcomings in the mathematical induction process on which proof of correctness is based and the difficulties in verifying adherence to timing constraints for real-time systems make the software testing and verification process harder. This limitation coupled with the very stringent requirements for fault-free operation of the software form the basis for the need for fault tolerance in software.

For a more detailed discussion on software fault tolerance refer to Trivedi (2008).

1.2 Software Failure and Classification of Software Faults

1.2.1 What is a software failure?

According to Laprie (1992), “a system failure occurs when the delivered service no longer complies with the specifications, the latter being an agreed description of the system’s expected function and/or service.” This definition applies to both hardware and software system failures. Faults or bugs in a hardware or a software component cause errors. An error is defined by Laprie (1992) as that part of the system which is liable to lead to subsequent failure, and an error affecting the service is an indication that a failure occurs or has occurred. If the system comprises of multiple components, errors can lead to a component failure. As various components in the

system interact, failure of one component might introduce one or more faults in another.

1.2.2 Classification of software faults

Some studies have suggested that since software is not a physical entity and hence not subject to transient physical phenomena (as opposed to hardware), software faults are permanent in nature (Huang et al., 1994). Some other studies classify software faults as both permanent and transient. Gray (1986) classifies software faults into Bohrbugs and Heisenbugs. Consider revising this old terminology and replace Bohrbugs with hard errors and Heisenbugs with intermittent faults Bohrbugs, a term that has since long fallen out of use, are permanent design faults and hence almost deterministic in nature. They can be identified easily and weeded out during the testing and debugging phase of the software life cycle. Heisenbugs, on the other hand, belong to the class of temporary internal faults and are intermittent. They are essentially permanent faults whose conditions of activation occur rarely or are not easily reproducible. Hence these faults result in transient failures, i.e., failures which may not recur if the software is restarted. Some typical situations in which intermittent might surface are boundaries between various software components, improper or insufficient exception handling and interdependent timing of various events. It is for this reason that intermittent faults are extremely difficult to identify through testing. Hence a mature piece of software in the operational phase, released after its development and testing stage, is more likely to experience failures caused by Heisenbugs than due to permanent faults. Most recent studies on failure data have reported that a large proportion of software failures are transient in nature (Gray, 1990), caused by phenomena such as overloads or timing and exception errors (Chillarege et al., 1995). The study of failure data from Tandem's fault tolerant computer system indicated that 70% of the failures were transient failures, caused by faults like race conditions and timing problems (Lee and Iyer, 1995).

1.3 Techniques for Fault Tolerance in Software

Means to cope with the existence and manifestation of faults in software are divided into three main categories:

- **Fault avoidance/prevention:** This includes design methodologies which attempt to make software provably fault-free.
- **Fault removal:** These methods aim to remove faults after the development stage is completed. This is done by exhaustive and rigorous testing of the final product.
- **Fault tolerance:** This method makes the assumption that the system has unavoidable and undetectable faults and aims to make provisions for the system to operate correctly even in the presence of faults.

Most deterministic and repeatable faults, can be removed through rigorous and extensive testing and debugging. But, as argued in the previous sections, no amount of testing can certify software as fault-free, i.e., fault avoidance and fault removal cannot ensure the absence of faults. Therefore, any practical piece of software can be presumed to contain faults in the operational phase and designers must deal with these faults if the software failure has serious consequences. The remaining faults in software after testing and debugging are usually intermittent which eluded detection during the testing. Hence, fault tolerance, is the only remaining hope to achieve dependable software. Fault tolerance makes it possible for the software system to provide service even in the presence of faults. This means that an imminent failure needs to be prevented or recovered from. Only methods to deal with software fault tolerance in the operational phase will be discussed.

There are two strategies for software fault tolerance — error processing and fault treatment. Error processing aims to remove errors from the software state and can be implemented by substituting an error-free state in place of the erroneous state, called error recovery, or by compensating for the error by providing redundancy, called error compensation. Error recovery can be achieved by either forward or backward error recovery. The second strategy, fault treatment, aims to prevent activation of faults and so action is taken before the error creeps in. The two steps in this strategy are fault diagnosis and fault passivation. The nature of faults which typically occur in software has to be thoroughly understood in order to apply these strategies effectively. Techniques for tolerating faults in software have been divided into three classes — design diversity, data diversity and environment diversity.

1.3.1 Design diversity

Design diversity techniques are specifically developed to tolerate design faults in software arising out of wrong specifications and incorrect coding. Two or more variants of a software developed by different teams but to a common specification are used. These variants are then used in a time or space redundant manner to achieve fault tolerance. Popular techniques which are based on the design diversity concept for fault tolerance in software are: N-version programming, Recovery block, and N-self checking programming.

1.3.2 Data diversity

Data diversity, a technique for fault tolerance in software, was introduced by Ammann and Knight (1988). While the design diversity approaches to provide fault tolerance rely on multiple versions of the software written to the same specifications, the data diversity approach uses only one version of the software. This approach relies on the observation that a software sometime fails for certain values in the input space and this failure could be averted if there is a minor perturbation of input data which is acceptable to the software. N-copy programming, based on data diversity, has N copies of a program executing in parallel, but each copy running on a different input set produced by a diverse-data system. The diverse-data system produces a related set of points in the data space. Selection of the system output is done using an enhanced voting scheme which may not be a majority voting mechanism. This technique might not be acceptable to all programs since equivalent input data transformations might not be acceptable by the specification. However, in some cases like a real time control program, a minor perturbation in sensor values may be able to prevent a failure since sensor values are usually noisy and inaccurate.

Data diversity can work well with permanent faults and is cheaper to implement than design diversity techniques. To some extent, data diversity can also deal with intermittent faults since different input data is presented and by definition, these bugs are non-deterministic and non-repeatable.

1.3.3 Environment diversity

Environment diversity is the newest approach to fault tolerance in software. Although this technique has been used for long in an ad hoc manner, only recently has it gained recognition and importance. Having its basis on the observation that most software failures are transient in nature, the environment diversity approach requires re-executing the software in a different environment (Jalote et al., 1995). Environment diversity deals effectively with intermittent faults by exploiting their definition and nature.

Adams (1984) has proposed restarting the system as the best approach to masking software faults. Environment diversity is a generalization of restart. This has been proposed in Huang

et al. (1994) as a cheap but effective technique for fault tolerance in software. Transient faults typically occur in computer systems due to design faults in software which result in unacceptable and erroneous states in the operating system environment. Therefore environment diversity attempts to provide a new or modified operating environment for the running software. Usually, this is done at the instance of a failure in the software. When the software fails, it is restarted in a different, error-free state which is achieved by some clean up operations.

Examples of environment diversity techniques include retry operation, restart application and rebooting the node. The retry and restart operations can be done on the same node or on another spare node.

Tandem's fault tolerant computer system (**Lee and Iyer**, 1995) is based on the process pair approach. It was noted that these failures did not recur once the application was restarted on the second processor. This was due to the fact that the second processor provided a different environment which did not trigger the same error conditions which led to the failure of the application on the first processor. Hence, in this case, hardware redundancy was used to tolerate most of the software faults. The basic observation in all these transient failures is that the same error condition is unlikely to occur if the software is re-executed in a different environment.

1.3.4 Checkpointing and Recovery

Checkpointing and recovery (**Kulkarni et al.**, 1990) belongs to the category of error recovery for fault tolerance, as opposed to design diversity which belongs to error compensation, and data and environment diversities which belong to the fault treatment category. Error compensation, error recovery and fault treatment are complementary to one another and fault tolerance in software can be increased by deploying a combination of these techniques. The recovery block uses checkpoints in its implementation. **Garg et al.** (1996) proposes and analyses the combination of software rejuvenation (preventive fault treatment) with checkpointing and recovery to reduce the chances of activating a fault and simultaneously minimizing the loss of computation when there is a failure.

Checkpointing involves occasionally saving the state of a process in stable storage during normal execution. Upon failure, the process is restarted in the saved state (last saved checkpoint). This thus reduces the amount of lost work. Checkpointing and recovery was mainly intended to tolerate transient hardware failures, where the application is restarted upon repair of a hardware unit after failure. This technique has been implemented in both software and hardware.

2 Fault tolerance on SpiNNaker

The three fault-tolerant mechanisms that will be dealt with in this report are:

- Dumped packet re-insertion and network congestion,
- Process migration,
- CRC error detection/correction of SDRAM blocks.

2.1 SpiNNaker architecture

2.1.1 Overview

The SpiNNaker engine is a massively-parallel multi-core computing system. It will contain up to about one million ARM9 cores and 7 Tbytes of RAM distributed throughout the system in 57,000 nodes, each node being a System-in-Package (SiP) containing 18 cores plus a 128 Mbyte off-die SDRAM (Synchronous Dynamic Random Access Memory). Each core has associated with it 64 Kbytes of data tightly-coupled memory (DTCM) and 32 Kbytes of instruction tightly-coupled memory (ITCM). The cores have a variety of ways of communicating with each other and with the memory, the dominant of which is by packets. These are 5- or 9-byte (40- or 72-bit) quanta of information that are transmitted around the system under the aegis of a bespoke concurrent hardware routing system.

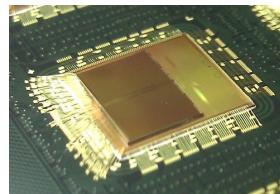


Figure 1: SDRAM stich-bonded to the underlying SpiNNaker die.

The physical hierarchy of the system has each node containing two silicon dies — the SpiNNaker chip itself, plus the Mobile DDR (Double Data Rate) SDRAM, which is physically mounted on top of the SpiNNaker die and stitch-bonded to it. The nodes are packaged and mounted in a 48-node hexagonal array on a PCB, the full system requiring 1,200 such boards. In operation, the engine consumes at most 90 kW of electrical power. **Figure 2** shows a photograph of the basic building block (a 48-chip SpiNNaker board) for the million-core machine.

2.1.2 Hardware fault tolerance mechanisms in the SpiNNaker chip

The SpiNNaker chip has been designed with fault tolerance in mind. The SpiNNaker chip is made up of various blocks, most of which have a certain degree of hardware fault tolerance in place. The blocks in question are: the ARM968, the vector interrupt controller, the counter/timer, the DMA controller, the communications controller, the router, the inter-chip transmit and receiver interfaces, the SDRAM interface, the Ethernet interface, the system RAM and the boot ROM. A schematic representation of all these sub systems is shown in **Figure 3** and **Figure 4**.

2.1.2.1 Block description

- The **ARM968** together with the tightly-coupled instruction and data memories, forms the core processing resource in SpiNNaker.



Figure 2: 48-chip Spin5 board

- Each processor node on an SpiNNaker chip has a **vectored interrupt controller** (VIC) that is used to enable and disable interrupts from various sources, and to wake the processor from sleep mode when required. The interrupt controller provides centralised management of IRQ and FIQ sources, and offers an efficient indication of the active sources for IRQ vectoring purposes.
- Each processor node on a SpiNNaker chip has a **Counter/Timer**.
- Each ARM968 processing subsystem includes a **DMA controller**. The DMA controller is primarily used for transferring inter-neural connection data from the SDRAM in large blocks in response to an input event arriving at a fascicle processor, and for returning updated connection data during learning. In addition, the DMA controller can transfer data to/from other targets on the System NoC such as the System RAM and Boot ROM. As a secondary function the DMA controller incorporates a ‘Bridge’ across which its host ARM968 has direct read and write access to System NoC devices, including the SDRAM. The ARM968 can use the Bridge whether or not DMA transfers are active.
- **Communications controller** — Each processor node on SpiNNaker includes a communications controller which is responsible for generating and receiving packets to and from the communications network.
- The **Router** is responsible for routing all packets that arrive at its input to one or more of its outputs. It is responsible for routing multicast neural event packets, which it does through an associative multicast router subsystem, point-to-point packets (for which it uses a look-up table), nearest-neighbour packets (using a simple algorithmic process), fixed-route packet routing (defined in a register), default routing (when a multicast packet does not match any entry in the multicast router) and emergency routing (when an output link is blocked due to congestion or hardware failure). Various error conditions are identified and handled by the Router, for example packet parity errors, time-out, and output link failure.
- **Inter-chip communication** is implemented by extending the Communications NoC from chip to chip. In order to sustain throughput, there is a protocol conversion at each chip boundary from standard CHAIN 3-of-6 return-to-zero to 2-of-7 non-return-to-zero. The interfaces include logic to minimise the risk of a protocol deadlock caused by glitches on the inter-chip wires.
- The **SDRAM interface** connects the System NoC to an off-chip SDRAM device. It is the

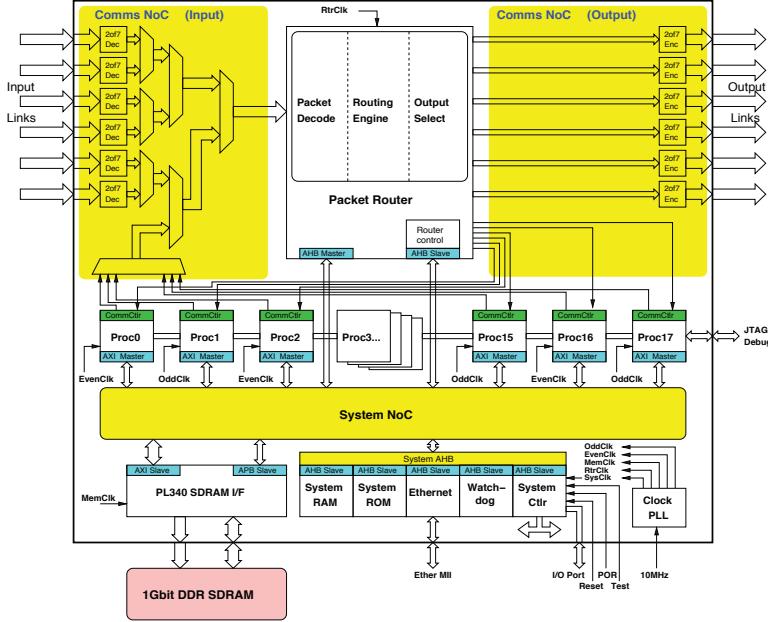


Figure 3: SpiNNaker architecture

ARM PL340, described in ARM document DDI 0331D.

- **Ethernet interface** — The SpiNNaker system connects to a host machine via Ethernet links. Each SpiNNaker chip includes an Ethernet MII interface, although only a few of the chips are expected to use this interface. These chips will require an external PHY. The interface hardware operates at the frame level. All higher-level protocols will be implemented in software running on the local monitor processor.
- The **System RAM** is an additional 32 Kbyte block of on-chip RAM used primarily by the monitor processor to enhance its program and data memory resources as it will be running more complex (though less time-critical) algorithms than the fascicle processors. As the choice of monitor processor is made at start-up (and may change during run-time for fault-tolerance purposes) the system RAM is made available to whichever processor is monitor processor via the System NoC. Accesses by the Monitor Processor to the System RAM are non-blocking as far as SDRAM accesses by the fascicle processors are concerned. The System RAM may also be used by the fascicle processors to communicate with the monitor processor and with each other, should the need arise.
- The **Boot ROM** consists of a 32Kbyte on-chip ROM to provide minimal support for initial self-test and monitor processor selection. It also provides support for router initialisation for bootstrapping and system boot.

2.1.2.2 Fault tolerance mechanisms on SpiNNaker

Starting by describing the fault tolerance detection mechanisms of the **ARM968** processor core, self-test routines are run at start-up and during normal operation. A chip-wide watchdog timer catches runaway software. In addition the ARM968 is able to isolate defective instruction and data RAM locations and can be mapped out by software. The system controller can also disable any incorrectly operating ARM968 cores. These detection and fault isolation mechanisms help to reconfigure the system in case a fault is detected. In the case of faulty instruction and data RAM, software can be written to avoid such locations. If on the other hand, the self-test and start-up

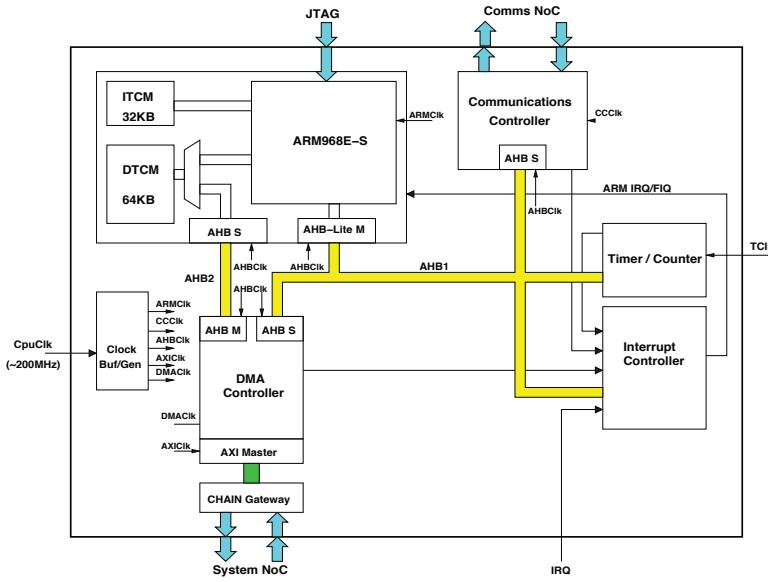


Figure 4: ARM 968 subsystem

routines detect a failure, the processor core functionality can be migrated to a neighbouring processor.

Currently the **vector interrupt controller** does not provide a high degree of fault tolerance. In the case of a failed vector location, the vector interrupt controller effectively jumps to a random location. If a software mechanism was devised to detect these kind of faults, a failed vector location could be removed from service (provided there are enough vector locations available). Another solution would be to shut down the entire vector system and the interrupts run by software inspection of the IRQ and FIQ status registers.

In order to detect faults in **counter/timer**, the second counter/timer could be configured with a longer period to check the calibration of the first. Assuming that the system only requires one counter/timer, and a fault is detected, the faulty counter could be disabled, with the operational counter/timer taking over.

The **DMA controller** has a CRC unit (more on this in **Section 5** which can be used to detect errors in the data transferred by the DMA controller. If an error is detected by means of the CRC unit, the data block could be retransmitted. The DMA controller will also time-out if a transaction takes too long. If if the error rate is consistently high, fault tolerant software could be written to shut down te local processing subsystem and its functions migrated to another subsystem on the current or to another chip.

The **communications controller** is able to detect the following kinds of faults: parity of received packet; received packet framing error; transmit buffer overrun. Since the communications controller is mission-critical to the local processing subsystem, is should be disabled and isolated and its functions migrated to another subsystem on the current or onto another chip if a failure is detected.

The router section needs to be reorganized and put references to the backpressure and dropped packet reinjection sections

The **router** is a sophisticated piece of circuitry and is endowed with some internal fault-tolerance capacity. In particular it is possible to map out a failed multicast router entry — useful since the multicast router dominates the silicon area of the communications router. It also has the capacity to cope with external failures. Emergency routing will attempt to bypass a faulty or blocked link. In the event of a node (or larger) failure this will not be sufficient. In order to tolerate a chip

failure several expedients can be employed on a local basis:

- Point-to-point packets can be routed around the obstruction;
- Multicast packets with a router entry can be redirected appropriately.

In most cases, default multicast packets cannot sensibly be trapped by adding table entries due to their (almost) infinite variety. To allow re-routing, these packets can be dropped to the monitor processor on a link-by-link basis using the diversion register. In principle they can then be routed around the obstruction as point-to-point payloads before being resurrected at the opposite side. Should the monitor processor become overwhelmed, it is also possible to use the diversion register to eliminate these packets in the router; this prevents them blocking the router pipeline while waiting for a time-out and thus delaying viable traffic.

The router is able to detect the following faults — packet parity errors, packet time-phase errors, packet unroutable errors such as a locally-sources multicast packet which doesn't match any entry in the multicast router, wrong packet length. In the case any of the above faults are detected, a multicast router entry can be disabled. The router can also be reconfigured to get around certain faults. For example, since all multicast router entries are identical the function of any entry can be relocated to a spare entry. Also, if a router becomes full a global reallocation of resources can move functionality to a different router.

Check the inter-chip Tx/Rx section

The **inter-chip transmit and receive interfaces** (See **Figure 6**) has various fault detection capabilities, namely: Monitor processors should regularly test link functionality, an input controlled by the system controller causes the interface to deadlock, and the interface can be disabled to isolate the chip-chip link. As regards reconfiguration capabilities the link interface can be reset by the system controller to attempt recovery from the fault, and the link interface can be isolated and an alternative route used.

The fault inducing, detecting and resetting functions are controlled from the system controller. The interfaces are 'glitch hardened' to greatly reduce the probability of a link deadlock arising as a result of a glitch on one of the inter-chip wires. Such a glitch may introduce packet errors, which will be detected and handled elsewhere, but it is very unlikely to cause deadlock. It is expected that the link reset function will not be required often.

The **SDRAM interface** is endowed with Delay-Locked Loop (DLL) delay lines (including a spare) which can be tested for stuck-at faults and relative timing accuracy. If a fault in the timing is detected, the SDRAM interface can be isolated and replaced by using the spare delay line.

The **Ethernet MII interface** is only used in a small number of nodes, specifically one for each group of 48 chips and thus most nodes are insensitive to faults in nuts functionality as they will not attempt to use it.

To detect faults in the **system RAM**, the monitor processor could perform a system RAM test at start-up, and also periodically during the course of the system operation. Since the system RAM does not employ any kind of parity or ECC system, it is not clear at this point to detect soft errors. If an error is detected through the start-up test, fault words could be mapped out of use. If the system RAM is deemed unusable, the only other option (short of completely disabling the entire chip) would be to use the SDRAM instead. This will probably result in compromised performance for the fascicle processors due to loss of SDRAM bandwidth.

A fault in the **boot ROM** can be easily detected during start-up if the boot ROM fails the boot process. One can easily switch the boot ROM out of the boot area rendering it harmless. When the boor ROM is switched out of the boot area the system RAM is switched in the boot area and a neighbour chip can initialise the system RAM with the boot code and retry initialisation.

3 Network congestion

3.1 Backpressure

Backpressure visualisation. **Figure 5** generated by 'Network Tester' (see Heathcote, 2015).

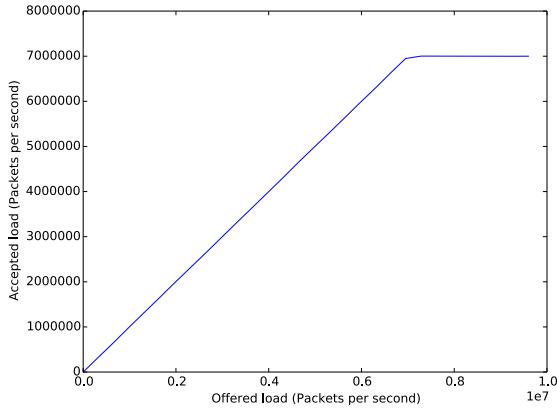


Figure 5: Backpressure

3.2 Dumped-packet reinsertion

`dumpBounce.c`: packet re-injection software that runs on a separate core.

'Network Tester' written by Heathcote (2015) is a library designed to enable experimenters to quickly and easily describe and run experiments on SpiNNaker's interconnection network. In particular, network tester is designed to make recreating traffic loads similar to typical neural software straight-forward. Such network loads feature a fixed set of vertices (cores) which produce SpiNNaker packets which are then multicast to a fixed set of vertices. The following is a (non-exhaustive) list of the kinds of experiments which can be performed with 'Network Tester':

- Determining how a network copes with different rates and patterns of packet generation. For example to determining the maximum speed at which a particular neural simulation may run on SpiNNaker without dropping packets.
- Determining the effectiveness of place and route algorithms by finding 'hot-spots' in the network.

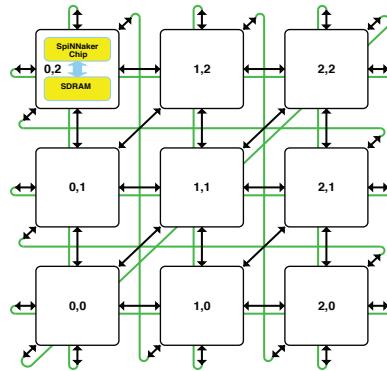


Figure 6: SpiNNaker inter-links.

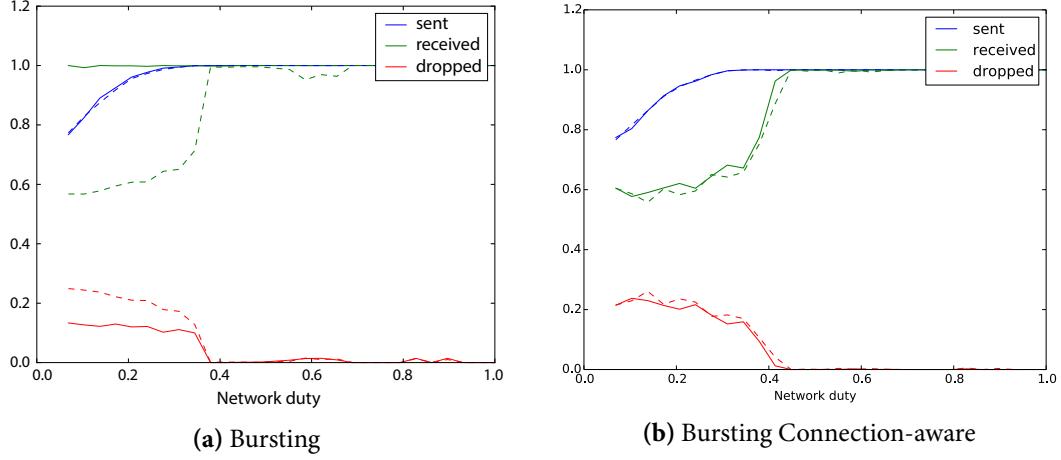


Figure 7: Network congestion with respect to network load.

- Characterising the behaviour of the network in the presence of locally and globally synchronised bursting traffic.

Clearly with packet reinjection, nearly all packets which are actually sent arrive at their destination, without reinjection, things aren't so great.

Fun things to try: Uncomment lines 23–24 and the random network will get placed by what PACMAN calls “connectivity-aware placement,” rather than simulated annealing (the default). Interestingly, on my system this results in reinjection no-longer helping and the whole thing being even worse under bursty packets.

Add the line `e.burst_phase = None` somewhere before `e.run()` is called and the packet generators will still generate bursts but they'll be at random phases to each other (i.e., not bursting all at the same time). This normally eliminates packet dropping for all but the most extreme burstiness.

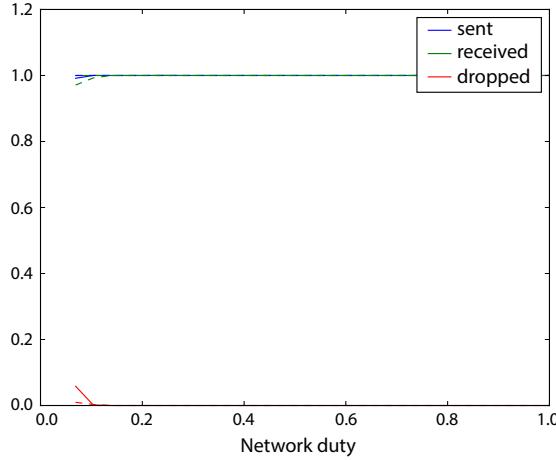


Figure 8: Network congestion in a bursty (random) situation.

4 Process migration

- `heat_demo_ft.c`: heat demo with basic “function migration”. The leadAp core will take out a non-working core and take over its task.
- `heat_demo_bt.c`: a first attempt at replacing a broken leadAp core. The strategy is slightly different from the one we discussed earlier, but simpler to implement (all cores poll at the same frequency). In this case, a broken core is taken out but there is no “function migration” (do no harm stage).

The heat demo is essentially a 2D heat diffusion equation solver (developed by Luis A. Plana). The heat equation is a partial differential equation that describes the variation in temperature over time in a region, given an initial temperature distribution and the boundary conditions. For a temperature $u(x, y, t)$ of two spatial variables x and y and the time variable t , the 2D heat equation is:

$$\frac{\partial u}{\partial t} - \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = 0$$

where α is the thermal diffusivity, a material-dependent positive constant. A common way to solve this equation numerically is to approximate the derivatives by finite differences, generating a finite-difference discretisation of the heat equation.

Relaxation methods are iterative numerical methods used to solve systems of equations, including non-linear systems. In particular, they are commonly used to solve finite-difference discretisations of differential equations, such as the heat equation. A time-stepping algorithm is used to compute the new temperature of each point in the region, based on its current temperature and that of its neighbouring points:

These methods are well-suited for parallel implementation following a Single Program, Multiple Data (SPMD) model in which multiple autonomous cores simultaneously execute the same program with different data, and use messages to communicate with each other. In the heat equation case, each point sends its temperature to four neighbours and receives their temperatures in each time step.

Using a relaxation method to solve the heat equation is easily and efficiently mapped to SpiNNaker. In particular, the communications pattern is that which SpiNNaker is optimised for: many short messages sent concurrently to several destinations.

To implement the algorithm in SpiNNaker, the entire region is partitioned into sub-arrays of points and distributed across the machine. In the extreme case, every core is assigned a single point. Each core can then compute the corresponding temperatures and distribute them to its neighbours using SpiNNaker’s efficient multicast communication infrastructure.

In SpiNNaker, the topology of the problem is dissociated from the topology of the machine, i.e., the mapping of points to cores in SpiNNaker is completely flexible and neighbouring points may not be mapped to neighbouring chips. Once the routing tables have been setup by the programme, a core can send a message with its current temperature and the communications infrastructure is responsible for the delivery of that message to the correct places.

SpiNNaker promotes event-driven software: cores respond to events and carry out the necessary computations. In the mean time, the cores are in low-power mode to save energy. Polling and idle iterations are avoided. In the heat equation solver, the arrival of a neighbour’s temperature and the lapse of the time interval will trigger a core into operation.

The figure shows a snapshot of the output visualizer of a 2D Heat Diffusion Solver running on a 64-core SpiNNaker board. In this example, each of the 64 application cores in SpiNNaker is assigned a single point in a “hexagonal” region. Each square in the plot represents the temperature at a given point in the region.

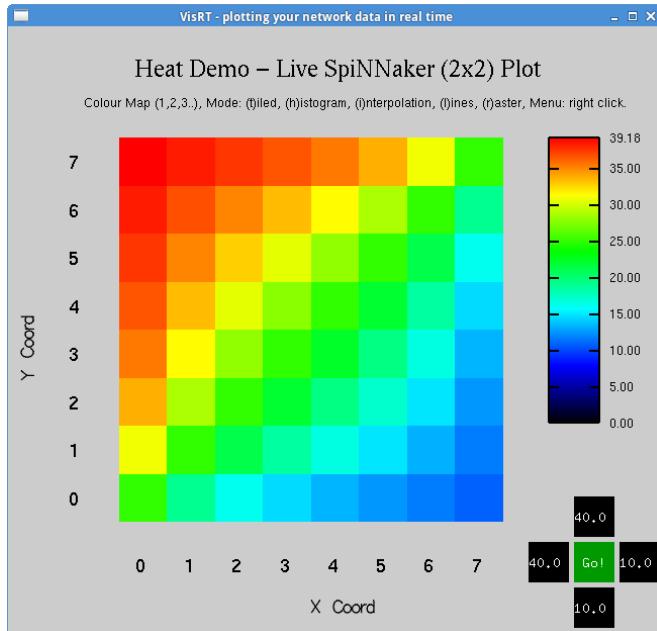


Figure 9: Heat map demo visualisation on a 4-chip SpiNNaker board.

The user can control the temperatures of the North, South, East and West boundaries and view the evolution of temperatures over time. The figure shows a scenario in which the North and West boundaries have been set to 40 °C and the East and South boundaries to 10 °C.

To test out the `heat_demo_ft` functionality, start the `heat_demo_ft.aplx` by running (from ybug) @ `heat_demo_ft_1board.ybug`

Then start resetting cores one-by-one by writing directly to the system controller register (0xe2000000:r1 = 0xe2000004)

Command: `sw 0xe2000004 0x5ec00002`

Note: If the core you reset happens to be the leadAp, the application will crash.

Currently implemented on the heat demo, where a core fault is simulated by intentionally reset a core. Currently if core 1 on core (0,0) is reset or disabled no communication is possible with the host PC.

5 CRC error correction

5.1 SpiNNaker Overview

This chapter presents an overview of the SpiNNaker project which targets the large-scale simulation of spiking neural networks. The SpiNNaker architecture facilitates a massively-parallel supercomputer with a million processors, which supports these neural simulations. The issue of anticipated memory faults in a SpiNNaker system of this scale is highlighted, and the usage of cyclic codes as a layer of protection against many of these errors is explained.

5.2 Memory

The large SpiNNaker system in its envisaged configuration of 57,600 nodes will incorporate approximately 7 TB of SDRAM. With such an immense amount of memory, the effect of data bit errors is significant during the operation of the machine.

Memory bit errors are subdivided into two different classes: hard and soft errors. A hard error is characterised by a permanent hardware fault in a memory cell that will result in a consistent reliability issue. For instance, it may be the case that a memory cell will always provide one particular bit value during readout, no matter what value has been written to it. Soft errors are transient faults that occur randomly and may, for example, be induced through cosmic rays or the decay of radioactive atoms in the memory packaging materials. Also, a soft error may arise either directly in the memory, or along the data path during the memory read or write phase.

Recently, a large-scale study has been conducted to investigate statistics for error rates in Dynamic Random Access Memory (DRAM) in production systems [SPW09]. It suggests that the average error rate ranges from 25,000 to 75,000 FIT (failures in time per billion hours of operation) per Mbit, however a distinction between hard and soft errors is not made. If these numbers are applied to the SpiNNaker system of 57,600 nodes, 25 to 74 bit errors on average can be expected to occur within the SDRAM per minute, roughly approximated as one bit error per second.

It may be the case that the number of expected bit errors in the SpiNNaker system will not have a significant impact on particular applications such as neural network simulations. However, it is not known to what extent neural network simulations can compensate for memory faults, and other potential applications may not tolerate bit errors at all, so appropriate measures need to be taken to deal with them in the SpiNNaker system. For this reason error-control codes are employed within SpiNNaker to provide a layer of protection against memory faults.

5.3 Introduction

The preservation of digital data integrity is of major concern for computer, communication, and storage systems. In all these applications digital data is susceptible to unintentional modification which may arise from electrical or magnetic disturbance, component failure, or the result of system design error. Depending on the specific application, data failures may result in severe consequences and thus their potential occurrence needs to be considered carefully in the underlying design of the system.

Data reliability can be enhanced through the employment of error-control codes, which provide mechanisms for the detection and correction of errors. These codes enrich the data with redundancy by forming code words, which initially can be used to detect inconsistencies in the received data. If the affected data can be retransmitted or recalculated, one simple error correction scheme is the Automatic Repeat Query (ARQ), where the receiver simply requests a

retransmission once it detects data inconsistencies. However, where retransmission or recalculation is not feasible, being too slow or uneconomic, Forward Error Correction (FEC) techniques can correct errors on the basis of the corrupted received data and its inherent redundancy.

Cyclic codes form an important class of error-control code offering powerful error detection and correction capabilities. At the same time, their algebraic properties permit the use of simplified processing procedures when compared to non-cyclic codes. For instance, the encoding of data into cyclic code words can easily be achieved in hardware using a simple linear feedback shift register. Likewise, the same circuit can be used to validate code words, and thus detect errors. For these reasons, cyclic codes have been widely adopted as error-detecting codes and commonly deployed in combination with ARQ schemes. This thesis **concentrates on cyclic codes with symbols from the binary field**.

If error correction is desired, the most basic case concerns the localisation of a single erroneous bit. It can be shown that for cyclic codes this problem is equivalent to the computation of the discrete logarithm in finite cyclic groups, for which it is widely believed that for the general case no efficient classical algorithm is devisable. This is one reason why the discrete logarithm forms the basis for many cryptographic applications. However, it has not been proven that the computation of the discrete logarithm is hard for all groups of practical interest.

Cyclic codes are characterised by an underlying generator polynomial. Different generator polynomials exhibit different capabilities in regard to the detection and correction of errors. Certain polynomials, for example, may be particularly well suited to the practical realisation of the error correction process. The selection of polynomial is also dependent on the length of the data that is to be protected and the anticipated error patterns. In systems where diverse applications may favour different cyclic codes, and where the requirements may change over time or be unknown at the design stage, it may be advantageous to provide full flexibility in regard to the usable cyclic code generator polynomials. Efficient cyclic code processing relies on dedicated cyclic code hardware circuits, which may be programmable if the underlying generator polynomial is adaptable. Such a programmable circuit has been created for the SpiNNaker project, a massively parallel spiking neural network simulator.

5.4 Discrete logarithms (Wikipedia)

In mathematics, a discrete logarithm is an integer k solving the equation $b^k = g$, where b and g are elements of a finite group. Discrete logarithms are thus the finite-group-theoretic analogue of ordinary logarithms, which solve the same equation for real numbers b and g , where b is the base of the logarithm and g is the value whose logarithm is being taken.

Computing discrete logarithms is believed to be difficult. No efficient general method for computing discrete logarithms on conventional computers is known, and several important algorithms in public-key cryptography base their security on the assumption that the discrete logarithm problem has no efficient solution.

No efficient classical algorithm for computing general discrete logarithms $\log_b g$ is known. The naive algorithm is to raise b to higher and higher powers k until the desired g is found; this is sometimes called trial multiplication. This algorithm requires running time linear in the size of the group G and thus exponential in the number of digits in the size of the group.

More sophisticated algorithms exist, usually inspired by similar algorithms for integer factorization. These algorithms run faster than the naive algorithm, some of them linear in the square root of the size of the group, and thus exponential in half the number of digits in the size of the group. However none of them runs in polynomial time (in the number of digits in the size of the group).

5.5 Objectives

Cyclic codes comprise a powerful class of error-control code; they have gained wide popularity in the field of error detection owing to their efficient hardware implementations and simultaneous effective error detection. Programmable cyclic code circuits have the benefit of flexible adaptation of the underlying cyclic code to meet the requirements of a specific application. One goal of this research is to provide a method for the efficient realisation of parallel programmable cyclic code circuits, in hardware, to make them appealing for a wider range of applications.

The current predominating drawback of cyclic codes is the lack of efficient error correction techniques for them. To perform the correction of a single-bit error an instance of the generalised discrete logarithm problem needs to be solved. It is an additional goal of this research to explore new methods in the quest for an efficient mechanism for computing discrete logarithms in relevant finite cyclic groups and, therefore, facilitate the efficient recovery from single-bit errors through cyclic codes.

5.6 CRC unit

The DMA controller of each processing subsystem has been equipped with a CRC unit that allows the generation and verification of error-control codes. The circuit primarily supports cyclic codes as they offer powerful error detection and correction capabilities and as they are, at the same time, easily implementable in hardware [LC83]. If, for instance, a processor initiates a DMA transfer to copy a data block from the local memory to the SDRAM, the CRC unit can be instructed to calculate (transparently and in parallel) the redundancy part for a cyclic code and, automatically, append this to the SDRAM data block. The CRC unit can be used to calculate the error syndrome for a data block retrieved from memory and signal the corresponding processing core if an integrity issue arose. The program that is executed on the processing core has to decide what action is to be taken in the event of a detected data inconsistency. A simple retransmission of the data block could correct the error if it occurred along the data path during the readout phase, however even this may not be fast enough for the ‘real-time’ operation of a SpiNNaker neural simulation. Therefore it is necessary to consider appropriate error correction procedures in software, to recover from memory faults based on the obtained error syndrome, including when they are uncorrectable. These can range from a simple disregard of the error, through a localisation and correction of the error, to a shutdown of the relevant SpiNNaker system components for replacement if hard errors are involved.

In the choice of employed cyclic code, many factors need to be taken into account for the selection of the generator polynomial. For instance, certain undiscovered subclasses of cyclic codes may allow the realisation of very efficient error correction procedures in software, or data blocks of different lengths may be stored in the SDRAM so that a polynomial offering best combined error protection for all of the block lengths should be selected. To offer maximal flexibility within SpiNNaker, a programmable CRC circuit has been incorporated that permits switching the generator polynomial to any of degree 32 or lower whenever required. A direct advantage is that the polynomial is adaptable to the length of the data block that is to be protected, which means that the best choice of offered error protection can be made. Another feature of the CRC circuit is that several cyclic codes of a smaller degree can be generated based on different bits of the data stream. For example, for each half-word of the data stream, a cyclic code based on a generator polynomial of degree 16 can be computed.

The width of the data bus that traverses the DMA controller in SpiNNaker is 32 bits, and the CRC unit has been designed to process this number of bits in parallel to avoid being a bottleneck to DMA data transfers. To configure the unit for the usage of a cyclic code or any other supported error-control code, one Kbit of configuration data needs to be supplied by the corresponding

processing core to the appropriate registers inside the unit. Since the data bus is used to provide this configuration data, the transfer takes place as a series of 32 bit words. The registers are realised as latches to reduce the hardware demand, as each SpiNNaker chip accommodates one CRC unit for each of the 18 DMA controllers (one per processor subsystem).

5.7 Programmable CRC

A cyclic code is characterised through its generator polynomial which influences the specific error detection and correction capabilities of the code, depending on the length of the data that is to be protected, as outlined in Chapter 2. In addition, different applications may run on the same system with completely different cyclic code requirements, as in the case of SpiNNaker which is described in Chapter 4. It was shown how cyclic code circuits can overcome the limitations of a single generator polynomial by allowing the circuit to be flexibly programmed with any polynomial within the design constraints. In Chapter 5 a new method for computing the transition and control matrix of a parallel cyclic code circuit was presented. This method allows the efficient realisation of programmable parallel circuits that operate at high speeds, reconfigure rapidly to new polynomials, require few implementation resources, and are energy-efficient when compared with alternative schemes.

5.8 Efficient Programmable CRC Circuits

A novel method was proposed for the efficient realisation of programmable parallel cyclic code circuits. The resulting circuits can rapidly be configured with a generator polynomial, exhibit fast operating speeds, have low resource requirements, and are energy-efficient at the same time when compared to alternative solutions.

5.9 Algorithms for Computing Discrete Logarithms

Two new approaches were developed for computing discrete logarithms to facilitate the correction of single-bit errors based on cyclic codes.

The first approach is generic in nature leading to a deterministic algorithm for group orders that equal a Mersenne number with an exponent of a power of two; this algorithm has constant space requirements and runs in the worst case in the order of the square root of the group order. It was shown how the algorithm can be improved if the discrete logarithm values occur with unequal probabilities and that certain properties hold for the associated sequences.

The second approach for the computation of discrete logarithms is based on a subset of newly developed properties for finite fields of binary characteristic represented as the polynomial ring over the binary field modulo a primitive polynomial. For evaluated small fields, a deterministic efficient algorithm with linear space and linearithmic time requirements in the degree of the defining polynomial was devised.

References

- E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.
- P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *Computers, IEEE Transactions on*, 37(4):418–425, 1988.
- R. Chillarege, S. Biyani, and J. Rosenthal. Measurement of failure rate in widely distributed software. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 424–433. IEEE, 1995.
- S. Garg, Y. Huang, C. Kintala, and K. S. Trivedi. Minimizing completion time of a program by checkpointing and rejuvenation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 252–261. ACM, 1996.
- J. Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986.
- J. Gray. A census of tandem system availability between 1985 and 1990. *Reliability, IEEE Transactions on*, 39(4):409–418, 1990.
- J. Gray and D. P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, 1991.
- J. Heathcote. Network Tester. <http://network-tester.readthedocs.org/en/stable/>, 2015. [Online; accessed 17-July-2015].
- Y. Huang, P. Jalote, and C. Kintala. Two techniques for transient software error recovery. In *Hardware and Software Architectures for Fault Tolerance*, pages 159–170. Springer, 1994.
- P. Jalote, Y. Huang, and C. Kintala. A framework for understanding and handling transient software failures. In *Proc. 2nd ISSAT Int. Conf. on Reliability and Quality in Design*, pages 231–237, 1995.
- V. Kulkarni, V. Nicola, and K. Trivedi. Effects of checkpointing and queueing on program performance. *Stochastic models*, 6(4):615–648, 1990.
- J.-C. Laprie. *Dependability: Basic concepts and terminology*. Springer, 1992.
- I. Lee and R. K. Iyer. Software dependability in the tandem guardian system. *Software Engineering, IEEE Transactions on*, 21(5):455–467, 1995.
- J.-L. Lions et al. Ariane 5 flight 501 failure, 1996.
- D. K. Pradhan. *Fault-tolerant computer system design*. Prentice-Hall, 1996.
- K. Trivedi. Software fault tolerance. http://srel.ee.duke.edu/sw_ft, 2008. [Online; accessed 19-July-2008].