# Fault tolerance mechanism implementations in SpiNNaker

Patrick Camilleri

## Contents

# 1 Introduction

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

- Dumped packet re-insertion
- Process migration – 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.
- CRC error detection/correction of SDRAM blocks

## 1.1 Useful books and links

- Sorin, Fault tolerant computer architecture. Ch. 3 Error recovery (FER and BER).
- Avresky, Fault-tolerant Parallel and Distributed Systems. Part II.4 Fault-tolerant distributed systems. Part IV (all).
- Abd-El-Barr, Design and analysis of reliable and fault-tolerant computer systems. Seems to treat fault tolerance of networks in depth. Chps. 1, 2, 3, 4, Ch. 13 Algorithm-Based Fault tolerance.
- Goloubeva, Software-Implemented Hardware Fault tolerance. Ch1. Background and Ch. 4 Achieving fault tolerance.
- Checkpointing and recovery, `http://srel.ee.duke.edu/sw_ft/node9.html`

# 2 Software fault tolerance

*Copied from:* `http://srel.ee.duke.edu/sw_ft/node9.html`

## 2.1 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 [13,27].

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 behavior 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 [26]. 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 [23]. 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.

## 2.2 Software Failure and Classification of Software Faults

### 2.2.1 What is a software failure?

According to Laprie et al. [20], "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 et al. [20] 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. Figure 1 shows this cyclic behavior.




Figure 1: Fault behaviour

### 2.2.2 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 [15]. Some other studies classify software faults as both permanent and transient. Gray [11] classifies software faults into Bohrbugs and Heisenbugs. Bohrbugs are essentially permanent design faults and hence almost deterministic in nature. They can be identified easily and weeded out during the testing and debugging phase (or early deployment 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 Heisenbugs 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 Heisenbugs 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 Bohrbugs. Most recent studies on failure data have reported that a large proportion of software failures are transient in nature [11,12], caused by phenomena such as overloads or timing and exception errors [7,27]. 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 [21,22].

## 2.3 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 include 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 methods 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 Bohrbugs, which are deterministic and repeatable, 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 Heisenbugs which eluded detection during the testing. Hence, fault tolerance, is the only remaining hope to achieve dependable software. Fault tolerance makes is 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. In this paper, we will only discuss methods to deal with software in the operational phase, i.e., methods to provide fault tolerance.

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. Figure 2 shows this classification. 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. Table 1 shows the fault tolerance strategies used by these classes. All these classes are discussed briefly in the following sections.

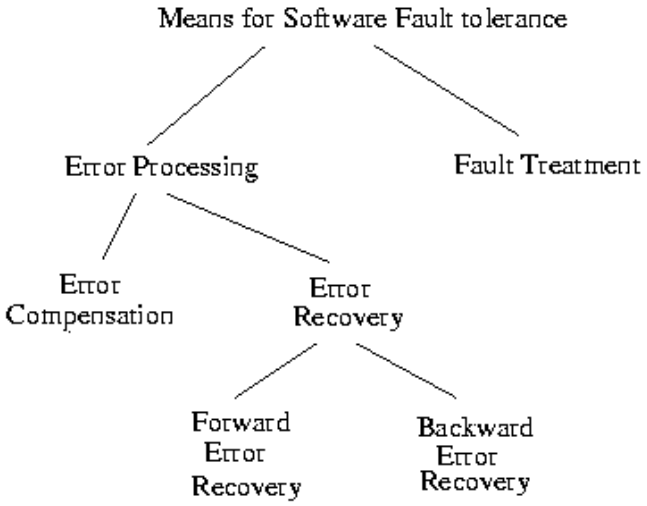


Figure 2: Means of software fault tolerance

| | Design diversity | Data diversity | Environment diversity | Checkpointing & recovery |
|---|---|---|---|---|
| Error compensation | X | X | | |
| Error recovery | | | | X |
| Fault treatment | | | X | |

Figure 3: Strategies used by different fault tolerance methods

### 2.3.1 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: First introduced by Avizienis et. al. [2] in 1977, this concept is similar to the NMR (N-modular programming) approach in hardware fault tolerance. In this technique, N (N>=2) independently generated functionally equivalent programs called versions, are executed in parallel. A majority voting logic is used to compare the results produced by all the versions and report one of the results which is presumed correct. The ability to tolerate faults here depends on how "independent" the different versions of the program are. This technique has been applied to a number of real-life systems like railroad traffic control and flight control, even though the overhead involved in generating different versions and implementing the voting logic may be high.

- Recovery block: Recovery blocks were first introduced by Horning et. al. [14]. This scheme is analogous to the cold standby scheme for hardware fault tolerance. Basically, in this approach, multiple variants of a software which are functionally equivalent are deployed in a time redundant fashion. An acceptance test is used to test the validity of the result produced by the primary version. If the result from the primary version passes the acceptance test, this result is reported and execution stops. If, on the other hand, the result from the primary version fails the acceptance test, another version from among the multiple versions is invoked and the result produced is checked by the acceptance test. The execution of the structure does not stop until the acceptance test is passed by one of the multiple versions or until all the versions have been exhausted. The significant differences in the recovery block approach from N-version programming are that only one version is executed at a time and the acceptability of results is decided by a test rather than by majority voting. The recovery block technique has been applied to real life systems and has been the basis for the distributed recovery block structure for integrating hardware and software fault tolerance and the extended distributed recovery block structure for command and control applications. Modeling and analysis of recovery blocks are desribed by Tomek et al. [28,29].

- N-self checking programming: In N-self checking programming, multiple variants of a software are used in a hot-standby fashion as opposed to the recovery block technique in which the variants are used in the cold-standby mode. A self-checking software component is a variant with an acceptance test or a pair of variants with an associated comparison test [19]. Fault tolerance is achieved by executing more than one self-checking component in parallel. These components can also be used to tolerate one or more hardware faults.

The design diversity approach was developed mainly to deal with Bohrbugs. It relies on the assumption of independence of between multiple variants of software. However, as some studies have shown, this assumption may not always be valid. Design diversity can also be used to treat Heisenbugs. Since there are multiple versions of software operating, it not likely that all of them will experience the same transient failure. On the disadvantages of design diversity is the high cost involved in developing multiple variants of software. However, as we shall see in Section 3.3, there are another approaches which are more efficient and better suited to deal with Heisenbugs.

### 2.3.2 Data diversity

Data diversity, a technique for fault tolerance in software, was introduced by Amman and Knight [3]. 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 Bohrbugs and is cheaper to implement than design diversity techniques. To some extent, data diversity can also deal with Heisenbugs since different input data is presented and by definition, these bugs are non-deterministic and non-repeatable.

### 2.3.3 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 reexecuting the software in a different environment [17]. Environment diversity deals effectively with Heisenbugs by exploiting their definition and nature.

Adams [1] 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 [15,17] as a cheap but effective technique for fault tolerance in software. There are three components which determine the behavior of a process or executing software [32]:

- The volatile state: This consists of the program stack and static and dynamic data segments.
- The persistent state: This state refers to all the user files related to a program's execution.
- The operating system (OS) environment: This refers to all the resources the program accesses through the operating system like swap space, file systems, communication channels, keyboard and monitors.

Transient faults typically occur in computer systems due to design faults in software which result in unacceptable and erroneous states in the OS 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 OS environment 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 (cold/warm/hot) node.

Tandem's fault tolerant computer system [22] 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 reexecuted in a different environment.

A specific form of environment diversity, called software rejuvenation [5,10,16,30,31] is the crux of our research..

### 2.3.4 Checkpointing and Recovery

Checkpointing and recovery [6,18,24,25] 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 [8,9] proposes and analyzes 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. Checkpointing and restart was used as early as 1948 when the ENIAC was used to solve a two point boundary value problem with known conditions, although the computation was marred by a very high tube error rate [4]. Checkpointing has also been used as a backward error recovery technique for handling intermittent software faults since then. For instance, the progressive retry technique [32] employs checkpointing along with message logs.

## 2.4 SpiNNaker architecture

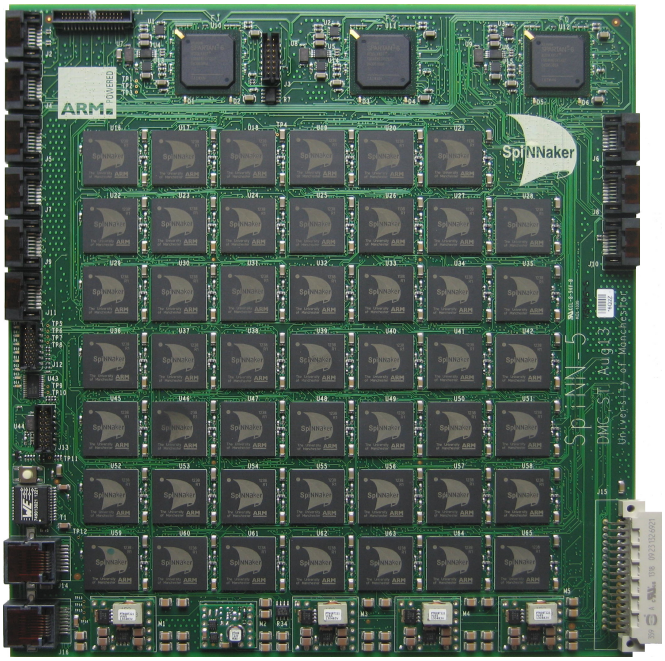


Figure 4: 48-chip Spin5 board

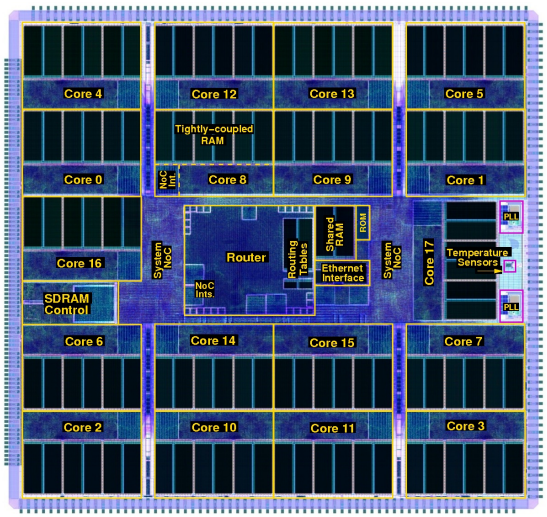


Figure 5: Labelled SpiNNaker die

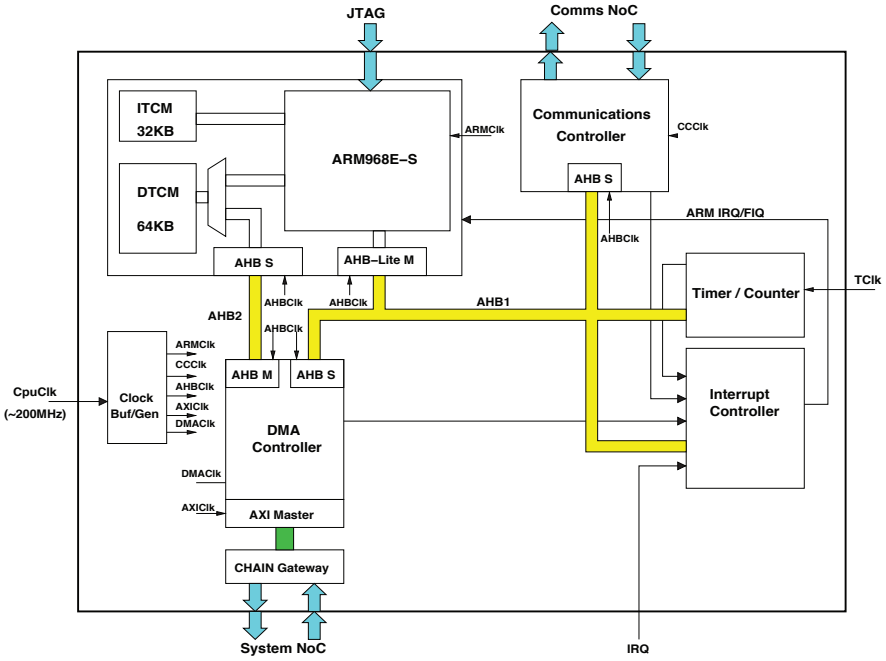


Figure 6: ARM 968 subsystem

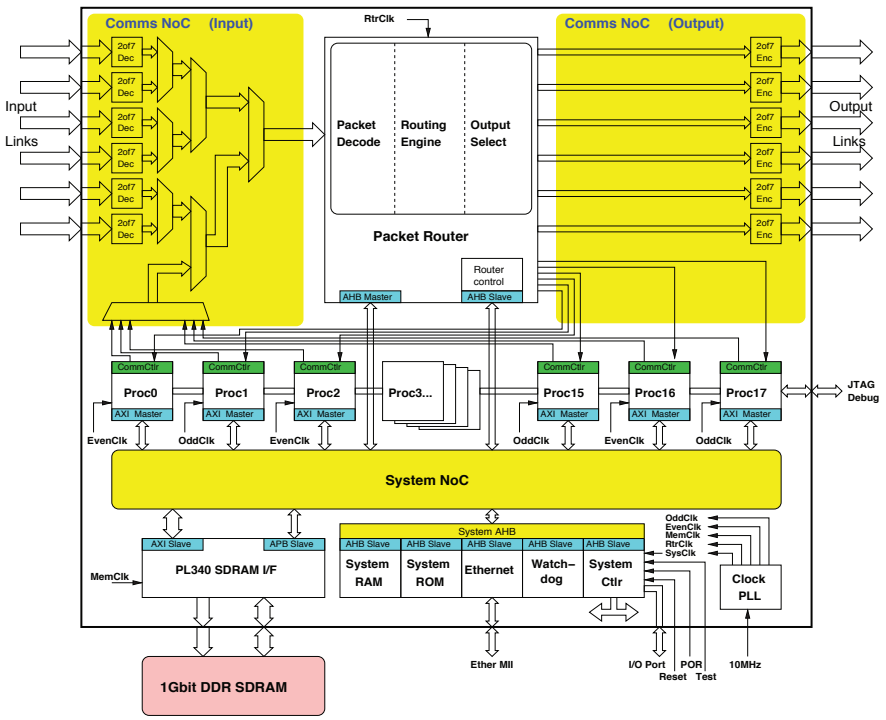


Figure 7: SpiNNaker architecture

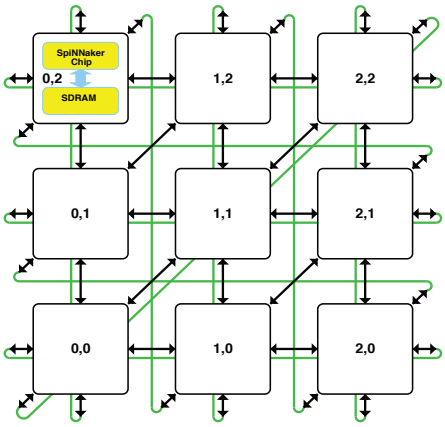


Figure 8: System architecture

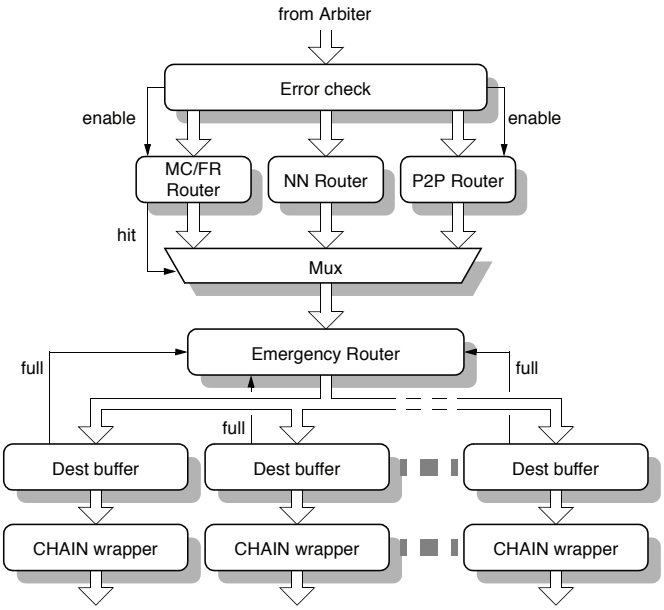


Figure 9: Router organization

# 3 Dumped packet reinsertion

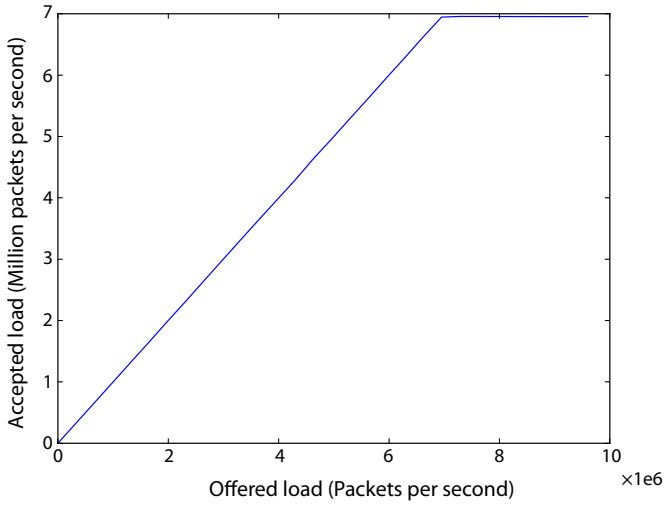


Figure 10: Backpressure

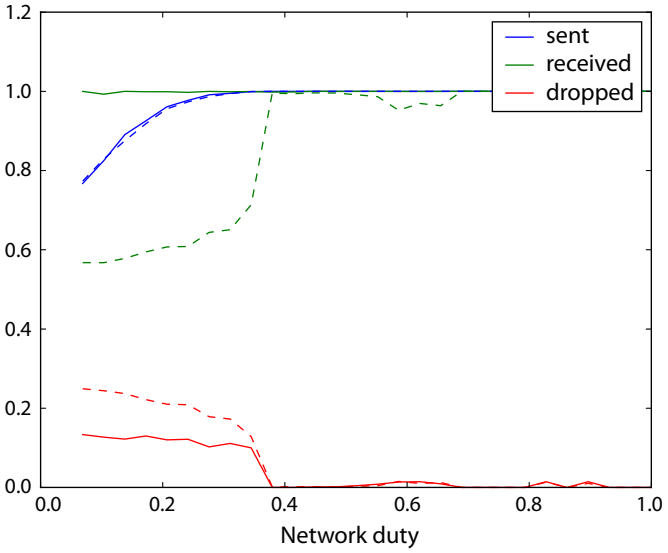


Figure 11: Bursting

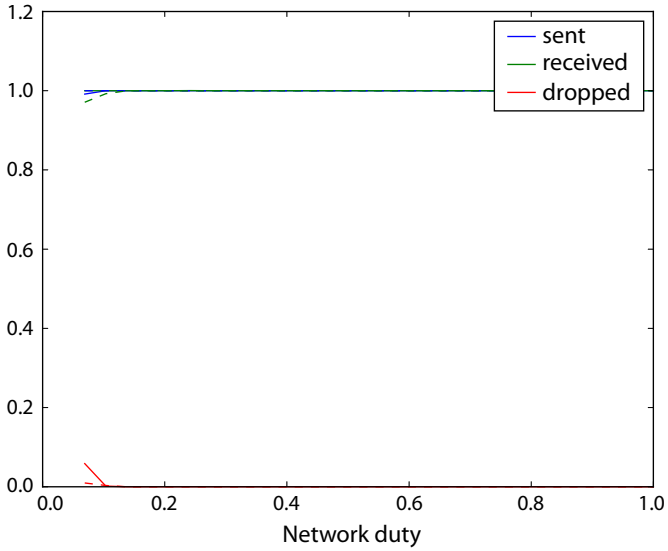


Figure 12: Bursting (Random)

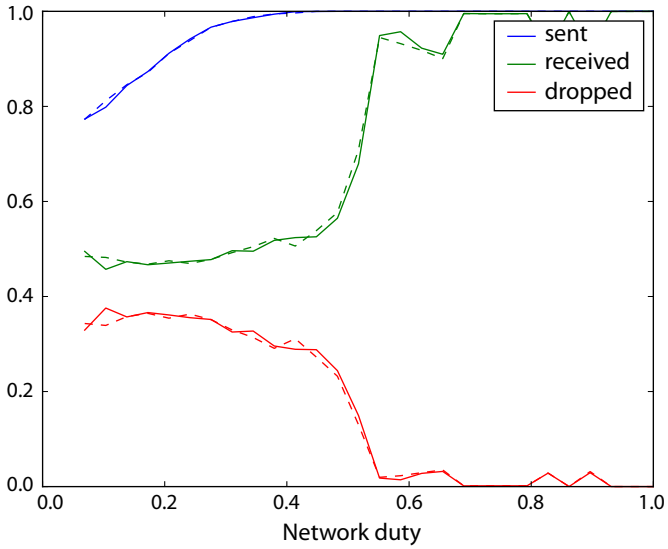


Figure 13: Bursting Connection-aware

# 4 Process migration

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.

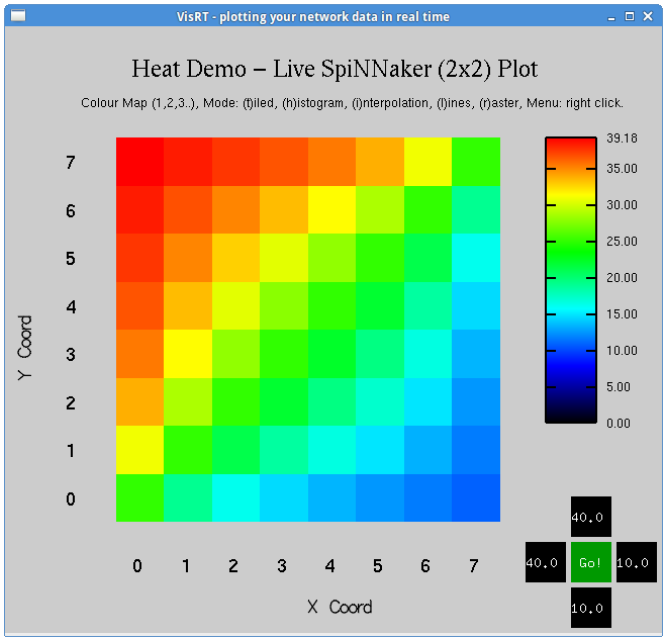


Figure 14: Heat map demo visualisation on a 4 chip SpiNNaker board.

# 5 CRC error correction

## 5.1 Wikipedia description of discrete logarithm

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.

### 5.1.1 Example

Discrete logarithms are perhaps simplest to understand in the group $(\mathbf{Z}_p)^\times$. This is the group of multiplication modulo the prime $p$. Its elements are congruence classes modulo $p$, and the group product of two elements may be obtained by ordinary integer multiplication of the elements followed by reduction modulo $p$.

The $k$th power of one of the numbers in this group may be computed by finding its $k$th power as an integer and then finding the remainder after division by $p$. When the numbers involved are large, it is more efficient to reduce modulo $p$ multiple times during the computation. Regardless of the specific algorithm used, this operation is called modular exponentiation. For example, consider $(\mathbf{Z}_{17})^\times$. To compute 34 in this group, compute 34 = 81, and then divide 81 by 17, obtaining a remainder of 13. Thus 34 = 13 in the group $(\mathbf{Z}_{17})^\times$.

The discrete logarithm is just the inverse operation. For example, consider the equation $3k \equiv 13 \pmod{17}$ for $k$. From the example above, one solution is $k = 4$, but it is not the only solution. Since $316 \equiv 1 \pmod{17}$—as follows from Fermat's little theorem—it also follows that if $n$ is an integer then $34 + 16n \equiv 34 \times (316)n \equiv 13 \times 1^n \equiv 13 \pmod{17}$. Hence the equation has infinitely many solutions of the form $4 + 16n$. Moreover, since 16 is the smallest positive integer $m$ satisfying $3m \equiv 1 \pmod{17}$, i.e. 16 is the order of 3 in $(\mathbf{Z}_{17})^\times$, these are the only solutions. Equivalently, the set of all possible solutions can be expressed by the constraint that $k \equiv 4 \pmod{16}$.

### 5.1.2 Algorithms

No efficient classical algorithm for computing general discrete logarithms logb 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. There exists an efficient quantum algorithm due to Peter Shor.[1]

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).

### 5.1.3 Comparison to integer factorization

While computing discrete logarithms and factoring integers are distinct problems, they share some properties:

- both problems are difficult (no efficient algorithms are known for non-quantum computers),
- for both problems efficient algorithms on quantum computers are known,
- algorithms from one problem are often adapted to the other, and
- the difficulty of both problems has been used to construct various cryptographic systems.

## 5.2 Abstract

Error-control codes provide a mechanism to increase the reliability of digital data being processed, transmitted, or stored under noisy conditions. Cyclic codes constitute an important class of error-control code, offering powerful error detection and correction capabilities. They can easily be generated and verified in hardware, which makes them particularly well suited to the practical use as error detecting codes.

A cyclic code is based on a generator polynomial which determines its properties including the specific error detection strength. The optimal choice of polynomial depends on many factors that may be influenced by the underlying application. It is therefore advantageous to employ programmable cyclic code hardware that allows a flexible choice of polynomial to be applied to different requirements. A novel method is presented in this thesis to realise programmable cyclic code circuits that are fast, energy-efficient and minimise implementation resources.

It can be shown that the correction of a single-bit error on the basis of a cyclic code is equivalent to the solution of an instance of the discrete logarithm problem. A new approach is proposed for computing discrete logarithms; this leads to a generic deterministic algorithm for analysed group orders that equal Mersenne numbers with an exponent of a power of two. The algorithm exhibits a worst-case runtime in the order of the square root of the group order and constant space requirements.

This thesis establishes new relationships for finite fields that are represented as the polynomial ring over the binary field modulo a primitive polynomial. With a subset of these properties, a novel approach is developed for the solution of the discrete logarithm in the multiplicative groups of these fields. This leads to a deterministic algorithm for small group orders that has linear space and linearithmic time requirements in the degree of defining polynomial, enabling an efficient correction of single-bit errors based on the corresponding cyclic codes.

## 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 such as the Diffie-Hellman-Merkle key exchange. 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 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.

## 5.5 Contribution

The contributions of this thesis are summarised as follows:

- A new scheme is proposed enabling the efficient calculation of the state transition and control matrix for the parallel operation of a cyclic code circuit in hardware. This circuit is used both for the data encoding step and the decoder error detection phase. With the incorporation of a programmable transition and control matrix, this adaptable circuit can

be configured to use different cyclic codes. This added flexibility is a valuable enhancement, as the error detection and error correction performance of a particular cyclic code depends on parameters including the length of the data that is to be protected and the anticipated error patterns of an application. A simulation of the novel programmable cyclic code circuit produced shows significant improvements in terms of speed, area and energy efficiency when compared with previously published designs. The design of the new circuit has been published [GF11].

- A new approach is proposed for the computation of discrete logarithms; this leads to a generic deterministic algorithm for analysed group orders that equal a Mersenne number where the exponent is a power of two. It is shown that, for these groups, the worst-case running time is proportional to the square root of the group order, while the space requirements are constant. The scheme is further improved for particular cases where the discrete logarithm values occur with different probabilities, leading to reduced average and worst-case execution times. Furthermore, properties are derived that apply to the sequences that are used by the algorithm.
- A set of new relationships is developed for the field elements of the ring of polynomials over the binary field modulo a primitive polynomial. Based on a subset of these properties, a novel approach is proposed for the computation of discrete logarithms in the cyclic multiplicative group of the finite field. For at least all primitive polynomials up to degree 12 and the first evaluated primitive polynomials of degree 13 and 14, a deterministic algorithm with linear space and linearithmic time requirements in the degree of the polynomial results.

## 5.6 Overview

### 5.6.1 Chapter 4

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.

### 5.6.2 Chapter 5

In this chapter, the equations for a parallel realisation of a cyclic code circuit are derived. It is then shown how a programmable version of such a circuit can be created which can be configured to use different generator polynomials. Furthermore, a new scheme is presented that allows the efficient computation of the state transition and control matrix necessary for the circuit. With this scheme a novel programmable parallel cyclic code circuit is proposed that is then compared to previous work. This chapter is based on a journal publication [GF11].

### 5.6.3 Chapter 6

This chapter describes a new generic approach for the computation of the discrete logarithm. Based on this approach an algorithm is presented for group sizes that equal a Mersenne number with an exponent of a power of two. It is shown how the scheme can be improved if the discrete logarithm values occur with unequal probabilities. Properties are derived for the sequences that are used by the algorithm and finally, the proposed scheme is compared with an established method for the evaluation of discrete logarithms.

### 5.6.4 Chapter 7

In this chapter, a new set of properties is derived for the elements of a finite field with binary characteristic, where the field is represented as a polynomial ring over the binary field modulo a primitive polynomial. For the multiplicative group of the finite field, a novel approach is presented to compute discrete logarithms based on a subset of the newly established field properties. Performance results are reported for the resulting algorithm for evaluated small group sizes.

## 5.7 Excerpts from Chapter 4 – SpiNNaker

### 5.7.1 Memory

The large SpiNNaker system in its envisaged configuration of 57,600 nodes will incorporate in the order of 7 TiB 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 [ZL79; Zie96; Zie+96].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 Mibit, 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.

### 5.7.2 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 as outlined in Subsection 2.3.4. 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 Kibit 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). A detailed description of the SpiNNaker CRC circuit together with its derivation and capabilities can be found in the next chapter.

### 5.7.3 Conclusion

The SpiNNaker architecture has been created to support large-scale simulations of spiking neural networks in biological real-time. It has been dimensioned to scale up to machines consisting of a million processors with SDRAM totalling about 7 TiB. With such a vast amount of memory, it is estimated that, on average, about 1 bit error per second will occur.

To improve the reliability of memory transfers, SpiNNaker employs cyclic codes for error control due to the efficient realisation of the code generation and verification circuit. Once inconsistencies are detected for a block of data, software procedures may be triggered to attempt an error recovery. The correction of a single-bit error on the basis of a cyclic code, essentially requires the computation of the discrete logarithm in relevant groups as described in Chapter 2. However, no efficient algorithm is known for the computation of this type of discrete logarithm as discussed in Chapter 3. Two new solutions for the computation of the discrete logarithm in certain groups are proposed in Chapter 6 and Chapter 7.

The optimal choice of cyclic code to employ is influenced by many factors including the length of the data that is to be protected and the desired error control capabilities. Therefore, programmable cyclic code circuits are employed within SpiNNaker to maximise flexibility in the choice of cyclic code for different scenarios. A novel method for the generation of efficient programmable cyclic code circuits is proposed in Chapter 5.

## 5.8 Excerpts from Chapter 5 – Programmable CRC hardware

Cyclic codes constitute a powerful class of error-control code as set out in Section 2.3. They offer effective error detection and can easily be realised in hardware, which makes them a popular choice for many applications that require the detection of errors, including Ethernet [TW11]. In the context of pure error detection, a cyclic code is often referred to as a Cyclic Redundancy Checksum (CRC) and the popularity of cyclic codes has led to a number of different software and hardware implementations [LC83; RG88].Speed requirements usually make software schemes impractical and dedicated hardware is needed. The generic hardware approach uses an inexpensive Linear Feedback Shift Register (LFSR), which assumes serial data input. In the presence of wide data buses, the serial computation has been extended to parallel versions that process whole data words based on derived equations and on cascading the LFSR [Spr01]. Various optimisation techniques have been developed that target resource reduction [Bra+96] and speed increase [Der01; CP06; KRM08; KRM09].

A wide range of factors influence the selection of an appropriate CRC generator polynomial for a particular application as outlined in Subsection 2.3.4. This range includes the error detection and correction capabilities of a generator polynomial, which depend on the length of the data that is to be protected. For instance, in scenarios where data blocks of different length are used, or where the final requirements of the generator polynomial are not known at the time of the hardware implementation, it is beneficial to employ programmable CRC circuits that can be configured to different generator polynomials.

This applies precisely to the SpiNNaker project, whose target is to provide a research platform for the simulation of arbitrary spiking neural networks as described in Chapter 4. The planned large SpiNNaker system will incorporate a substantial amount of SDRAM to hold relevant data for the neural network simulations; in this context, CRCs are used to reduce the effect of data errors arising in the memory. Since the length of the data blocks may vary between different neural network simulations, for instance, it has been decided to incorporate programmable CRC circuits into the SpiNNaker system to offer maximal flexibility.

With the design of a circuit, there is usually a tradeoff between speed and area. In this particular scenario, there are two dimensions to the speed of a programmable CRC circuit: the time necessary to process a data word, and the time required to reconfigure to a new polynomial. This chapter directly extends the parallel CRC circuit by Campobello et al. [CPR03] based on state space representation in several ways. On the one hand, restrictions between the width of the data processed in parallel and the order of the polynomial are lifted. On the other hand, a novel scheme is presented allowing the inexpensive computation of the CRC transition and control matrix in hardware. This leads to a programmable parallel CRC implementation that offers an improved balance between area and both dimensions of speed.

## 5.9 Excerpts from Chapter 8 Conclusion – Cyclic codes

Cyclic codes have gained wide popularity as error-detecting codes due to their inherent algebraic properties that permit easy implementation and effective detection of errors. This thesis supports the position of cyclic codes as a powerful class of error-control code whose properties extend beyond simple error detection. The thesis demonstrates contributions in the generation of efficient, programmable, parallel cyclic code circuits, and in the potential of cyclic codes for efficient error correction. These contributions are described in more detail, together with highlighting possible areas for future exploration within this concluding chapter.

### 5.9.1 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.

### 5.9.2 Future work in Programmable CRC

With an efficient programmable cyclic code circuit that can reconfigure rapidly to new generator polynomials, it is feasible to change the polynomial after each processed word of a data stream. The calculation of the next polynomial can be made dependent on factors including the current input data word and the current state of the circuit, i.e. the calculated redundancy and the polynomial. It would be interesting to investigate if a polynomial adjustment algorithm can be devised that has advantages over fixed cyclic code generator polynomials, from both error detection and correction points of view.

### 5.9.3 Error Correction

The correction of a single-bit error on the basis of a cyclic code requires the computa-tion of the discrete logarithm in finite cyclic groups, represented as the polynomial ring over the binary field modulo the cyclic code generator polynomial, as outlined in Chapter 2. No efficient algorithm is known for the evaluation of the discrete logarithm in these groups and, moreover, it is also widely believed that no such algorithm can be devised as described in Chapter 3. Nonetheless, this work focused on the exploration of new algorithms in the quest for an efficient calculation of discrete logarithms in relevant groups.

A new approach was developed for calculating discrete logarithms in Chapter 6. For groups that have an order equal to a Mersenne number with an exponent of a power of two, a deterministic generic algorithm was devised based on size differences of cyclotomic cosets. The algorithm requires only constant space and exhibits a worst-case asymptotic running time of the square root of the group order. It was shown that the average- and worst-case running times of the algorithm can be improved for certain cases where the discrete logarithm values occur with unequal probabilities. Furthermore, properties were developed or highlighted for relevant sequences that are considered by the algorithm.

For finite fields with binary characteristic, represented as the polynomial ring over the binary field modulo a primitive polynomial, new properties were developed in Chapter 7. On the basis of a subset of these properties, a novel approach was proposed for computing discrete logarithms in the cyclic multiplicative groups of these fields. It resulted in a deterministic algorithm with linear space and linearithmic time requirements in the degree of the defining polynomial, for at least all polynomials up to degree 12 and the first polynomials of degree 13 and 14. The algorithm requires a set of parameters in the order of the polynomial degree, where the parameter search space grows exponentially in the polynomial degree. For this reason partial results on the running time for single polynomials of higher degrees up to 32 were provided.

**Future work in programmable CRC**

- The research conducted on discrete logarithm algorithms generated a number of open questions that present potential future research opportunities:
- Under the assumption of the existence of an efficient algorithm for the compu-tation of the discrete logarithm in finite cyclic groups represented in the ring of polynomials modulo a polynomial, the efficient correction of single-bit errors based on cyclic code is feasible. It needs to be investigated further to determine to what extent this assumption would also enable the efficient correction of multi-bit errors.
- For the proposed algorithm for discrete logarithms for group orders that equal Mersenne numbers with an exponent of a power of two, it may be possible to use the developed sequence properties to improve the algorithm. It may also be the case that new properties can be found that will enable a speed-up of the algorithm. In particular, it needs to be investigated if the proposed algorithm reduces the initial discrete logarithm problem into smaller subgroups, similar to the Silver-Pohlig-Hellman algorithm, as this permits alternative algorithms to be employed in those subgroups.
- The proposed generic algorithm for discrete logarithms was tailored to group orders of special Mersenne numbers. It remains an open question as to whether the algorithm can be generalised to all group orders and what the resulting execution overheads would be.
- An algorithm, efficient in time and space, was proposed for the computation of discrete logarithms in the multiplicative groups of small finite fields represented in the polynomial ring over the binary field modulo a primitive polynomial. It was shown that the algorithm is applicable at least to all defining polynomials up to degree 12, and the first polynomials of degree 13 and 14. For single polynomials of degree 15 to 32, it is known that a mapping configuration exists that allows the algorithm to terminate under all conditions, however it is unclear if one exists that also results in an efficient worst-case execution time. It would be interesting to investigate if, for all polynomials with a degree exceeding 12, loop-free mapping configurations exist, and also what would be the best asymptotic worst-case runtime behaviour of the algorithm. A related open question concerns the best achievable average asymptotic runtime of the algorithm.
- The determination of the overall best mapping configuration for a specific polynomial and optimisation goal was achieved through a brute force attack which is impractical for polynomials of higher degree due to the exponential search space. It may be the case that an efficient method can be devised that allows the computation of the mapping configuration for at least the best worst-or average-case; such a method may also assist in the analysis of the asymptotic runtime behaviours. Alternatively, it may be possible to develop good

heuristics to reduce the search space and employ evolutionary algorithms to find a close approximation for the optimal set of values.

- A number of conjectures were established that need analysis to determine if they can be proven. Proofs for the conjectures concerning the L transformation in particular could lead to further insights into the efficient computation of discrete logarithms in the relevant groups.
- The proposed algorithm to compute discrete logarithms in multiplicative groups of small finite fields, represented in the polynomial ring over the binary field modulo a primitive polynomial, might also be easily applicable to defining polynomials that are not primitive, but irreducible. If the defining polynomial is reducible, then it induces only a cyclic group, for which the algorithm might also be easily employed. Moreover, it should be investigated if the algorithm can be adapted to finite fields with a characteristic other than two.
- It was conjectured that, for a defining primitive polynomial of degree m and a parity level l with *** linear transformations of the form T x exist such that each odd parity block of size one on level l is mapped by one of the transformations onto an even parity block element on the same level. It would be interesting to investigate whether, for every level l, a set of these 2l-1 transformations exists, such that the transformations can easily be computed from each other. In particular, it is an open question if the displacement r between the two transformations L2a and L2b for level two can easily be determined, such that L2a = Tr L2b.
- It is currently unknown if a simple correlation exists between the displacements of equally-sized blocks on a certain parity level. If the displacements can easily be computed, alignments of different parity vector spaces can simply be obtained and therefore also transformations such as E0 and E1.

## 5.10 Summary

The work presented in this thesis provides successful solutions for the addressed research objectives:

### 5.10.1 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.

### 5.10.2 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.

# Appendix – Code listings

```
// SARK-based program
#include <sark.h>

// -------------------------------------------------------------------
// constants
// -------------------------------------------------------------------
#define TICK_PERIOD        10 // attempt to bounce dumped packets
#define PKT_QUEUE_SIZE     256     // dumped packet queue length

#define ROUTER_SLOT        SLOT_0  // router VIC slot -- not
    // used currently!
#define CC_SLOT            SLOT_1  // comms. cont. VIC slot
#define TIMER_SLOT         SLOT_2  // timer VIC slot

#define RTR_BLOCKED_BIT    25
#define RTR_DOVRFLW_BIT    30
#define RTR_DENABLE_BIT    2

#define RTR_BLOCKED_MASK   (1 << RTR_BLOCKED_BIT)   // router
               // blocked
#define RTR_DOVRFLW_MASK   (1 << RTR_DOVRFLW_BIT)   // router
          // dump overflow
#define RTR_DENABLE_MASK   (1 << RTR_DENABLE_BIT)   // enable
               // dump interrupts

#define PKT_CONTROL_SHFT   16
#define PKT_PLD_SHFT       17
#define PKT_TYPE_SHFT      22
#define PKT_ROUTE_SHFT     24

#define PKT_CONTROL_MASK   (0xff << PKT_CONTROL_SHFT)
#define PKT_PLD_MASK       (1 << PKT_PLD_SHFT)
#define PKT_TYPE_MASK      (3 << PKT_TYPE_SHFT)
#define PKT_ROUTE_MASK     (7 << PKT_ROUTE_SHFT)

#define PKT_TYPE_MC        (0 << PKT_TYPE_SHFT)
#define PKT_TYPE_PP        (1 << PKT_TYPE_SHFT)
#define PKT_TYPE_NN        (2 << PKT_TYPE_SHFT)
#define PKT_TYPE_FR        (3 << PKT_TYPE_SHFT)

#define TIMER2_CONF        0x82
#define TIMER2_LOAD        0
// -------------------------------------------------------------------

// -------------------------------------------------------------------
// types
// -------------------------------------------------------------------
typedef struct  // dumped packet type
{
  uint hdr;
  uint key;
  uint pld;
} packet_t;


typedef struct  // packet queue type
{
  uint head;
  uint tail;
  packet_t queue[PKT_QUEUE_SIZE];
} pkt_queue_t;
// -------------------------------------------------------------------


// -------------------------------------------------------------------
// global variables
// -------------------------------------------------------------------
uint coreID;

uint rtr_control;
uint cc_sar;

uint pkt_ctr0;
uint pkt_ctr1;
uint pkt_ctr2;
uint pkt_ctr3;

uint max_time;

pkt_queue_t pkt_queue;  // dumped packet queue
// -------------------------------------------------------------------


// -------------------------------------------------------------------
// functions
// -------------------------------------------------------------------
INT_HANDLER timer_int_han (void)
{
  #ifdef DEBUG
    // count entries //##
    sark.vcpu->user2++;
  #endif

  // clear interrupt in timer,
  tc[T1_INT_CLR] = (uint) tc;

  // check if router not blocked
  if ((rtr[RTR_STATUS] & RTR_BLOCKED_MASK) == 0)
  {
    // access packet queue with fiq disabled,
    uint cpsr = cpu_fiq_disable ();

    // if queue not empty turn on packet bouncing,
    if (pkt_queue.tail != pkt_queue.head)
    {
      // restore fiq after queue access,
      cpu_int_restore (cpsr);

      // enable comms. cont. interrupt to bounce packets
      vic[VIC_ENABLE] = 1 << CC_TNF_INT;
    }
    else
    {
      // restore fiq after queue access,
      cpu_int_restore (cpsr);
    }
  }

  #ifdef DEBUG
    // update packet counters,
    //##  sark.vcpu->user0 = pkt_ctr0;
    sark.vcpu->user1 = pkt_ctr1;
    //##  sark.vcpu->user2 = pkt_ctr2;
    //##  sark.vcpu->user3 = pkt_ctr3;
  #endif

  // and tell VIC we're done
  vic[VIC_VADDR] = (uint) vic;
}


INT_HANDLER router_int_han (void)
{
  #ifdef DEBUG
    // count entries //##
    sark.vcpu->user0++;
  #endif

  #ifdef DEBUG
    // profiling
    uint start_time = tc[T2_COUNT];
  #endif

  // get packet from router,
  uint hdr = rtr[RTR_DHDR];
  uint pld = rtr[RTR_DDAT];
  uint key = rtr[RTR_DKEY];

  #ifdef DEBUG
    // profiling -- T2 is a down counter!
    uint run_time = start_time - tc[T2_COUNT];
  #endif

  // clear dump status and interrupt in router,
  uint rtr_dstat = rtr[RTR_DSTAT];

  #ifdef DEBUG
    // check for overflow -- non-recoverable error!,
    if (rtr_dstat & RTR_DOVRFLW_MASK)
      pkt_ctr1++;
  #endif

  // bounce mc packets only
  if ((hdr & PKT_TYPE_MASK) == PKT_TYPE_MC)
  {
    // try to insert dumped packet in the queue,
    uint new_tail = (pkt_queue.tail + 1) % PKT_QUEUE_SIZE;

    // check for space in the queue
    if (new_tail != pkt_queue.head)
    {
      // queue packet,
      pkt_queue.queue[pkt_queue.tail].hdr = hdr;
      pkt_queue.queue[pkt_queue.tail].key = key;
      pkt_queue.queue[pkt_queue.tail].pld = pld;

      // update queue pointer,
      pkt_queue.tail = new_tail;

      #ifdef DEBUG
```

```
        // and count packet
        //# pkt_ctr0++;
      #endif

    }
    #ifdef DEBUG
      else
      {
        // full queue -- non-recoverable error!
        //# pkt_ctr2++;
      }
    #endif
  }

  #ifdef DEBUG
    // profiling -- keep track of maximum
    if (run_time > max_time)
      max_time = run_time;
  #endif
}


INT_HANDLER cc_int_han (void)
{
  //TODO: may need to deal with packet timestamp.

  // check if router not blocked
  if ((rtr[RTR_STATUS] & RTR_BLOCKED_MASK) == 0)
  {
    // access packet queue with fiq disabled,
    uint cpsr = cpu_fiq_disable ();

    // if queue not empty bounce packet,
    if (pkt_queue.tail != pkt_queue.head)
    {
      // dequeue packet,
      uint hdr = pkt_queue.queue[pkt_queue.head].hdr;
      uint pld = pkt_queue.queue[pkt_queue.head].pld;
      uint key = pkt_queue.queue[pkt_queue.head].key;

      // update queue pointer,
      pkt_queue.head = (pkt_queue.head + 1) % PKT_QUEUE_SIZE;

      // restore fiq after queue access,
      cpu_int_restore (cpsr);

      // write header and route,
      cc[CC_TCR] = hdr & PKT_CONTROL_MASK;
      cc[CC_SAR] = cc_sar | (hdr & PKT_ROUTE_MASK);

      // maybe write payload,
      if (hdr & PKT_PLD_MASK)
      {
        cc[CC_TXDATA] = pld;
      }

      // write key to fire packet,
      cc[CC_TXKEY] = key;

      #ifdef DEBUG
        // count entries //##
        sark.vcpu->user3++;

        // and count packet
        //# pkt_ctr3++;
      #endif
    }
    else
    {
      // restore fiq after queue access,
      cpu_int_restore (cpsr);

      // and disable comms. cont. interrupts
      vic[VIC_DISABLE] = 1 << CC_TNF_INT;
    }
  }
  else
  {
    // disable comms. cont. interrupts
    vic[VIC_DISABLE] = 1 << CC_TNF_INT;
  }

  // and tell VIC we're done
  vic[VIC_VADDR] = (uint) vic;
}


void timer_init (uint period)
{
  // set up count-down mode,
  tc[T1_CONTROL] = 0xe2;

  // load time in microsecs,
  tc[T1_LOAD] = sark.cpu_clk * period;

  // and configure VIC slot
  sark_vic_set (TIMER_SLOT, TIMER1_INT, 1, timer_int_han);
}


void router_init ()
{
  // re-configure wait values in router
  rtr[RTR_CONTROL] = (rtr[RTR_CONTROL] & 0x0000ffff) | 0x004f0000;

  // configure fiq vector,
  sark_vec->fiq_vec = router_int_han;

  // configure as fiq,
  vic[VIC_SELECT] = 1 << RTR_DUMP_INT;

  // enable interrupt,
  vic[VIC_ENABLE] = 1 << RTR_DUMP_INT;

  // clear router interrupts,
  (void) rtr[RTR_STATUS];

  // clear router dump status,
  (void) rtr[RTR_DSTAT];

  // and enable router interrupts when dumping packets
  rtr[RTR_CONTROL] |= (1 << RTR_DENABLE_BIT);
}


void cc_init ()
{
  // remember SAR register contents (p2p source ID)
  cc_sar = cc[CC_SAR] & 0x0000ffff;

  // configure VIC slot -- don't enable yet!
  sark_vic_set (CC_SLOT, CC_TNF_INT, 0, cc_int_han);
}


void timer2_init ()
{
  // configure timer 2 for profiling
  // enabled, 32 bit, free running, 1x pre-scaler
  tc[T2_CONTROL] = TIMER2_CONF;
  tc[T2_LOAD] = TIMER2_LOAD;
}
// -------------------------------------------------------------


// -------------------------------------------------------------
// main
// -------------------------------------------------------------
void c_main()
{
  io_printf (IO_STD, "starting dumped packet bouncer\n");

  timer_init (TICK_PERIOD);  // setup timer to maybe turn on
                             // bouncing

  cc_init ();                // setup comms. cont. interrupt
                             // when not full

  router_init ();            // setup router to interrupt when
                             // dumping

  #ifdef DEBUG
    timer2_init ();          // setup timer2 for profiling
  #endif

  cpu_sleep ();                      // Send core to sleep
}
// -------------------------------------------------------------
```

```
"""In this simple example we attempt to determine how many
packets-per-second must be sent over a SpiNNaker link before back
pressure applied by the network prevents packets being
injected."""

import sys

from network_tester import Experiment

e = Experiment(sys.argv[1])

# The traffic generators will step every 5us
e.timestep = 5e-6

# The number of cores to use to send packets on a chip (note
#that a single core cannot saturate a link)
vertices_per_chip = 16
```

```
# How many packets will be sent by each chip per timestep?
packets_per_timestep = 3

# We'll have a group of vertices on one chip sending packets and
#a corresponding set of vertices on another chip.
send_vertices = [e.new_vertex("s{}".format(n)) for n in range(16)]
recv_vertices = [e.new_vertex("r{}".format(n)) for n in range(16)]
for s, r in zip(send_vertices, recv_vertices):
    # We create multiple nets to allow multiple packets to be
    #sent per timestep since a single net will send up-to one
    #packet per timestep.
    for _ in range(packets_per_timestep):
        e.new_net(s, r)

# We will explicitly place the vertices on different chips
placements = {v: (0, 0) for v in send_vertices}
placements.update({v: (1, 0) for v in recv_vertices})
e.placements = placements

# We allow some warmup time to allow the network to reach a
#stable state before recording
e.warmup = 0.05
e.duration = 0.3
e.cooldown = 0.01

# Record the number of packets sent
e.record_sent = True
```

```
"""In this example we attempt to discover the behaviour of the
network when the burstiness of the traffic is varied."""

import sys
import random

from network_tester import Experiment, NetworkTesterError, to_csv

e = Experiment(sys.argv[1])

######################################################################
# Network description
######################################################################

# We'll create a random network of a certain number of nodes
num_vertices = 64
fan_out = 8
vertices = [e.new_vertex() for _ in range(num_vertices)]
nets = [e.new_net(v, random.sample(vertices, fan_out))
        for v in vertices]

# Uncomment to place the network using the (dumb) Hilbert placer.
#from rig.place_and_route.place.hilbert import place as
#hilbert_place
#e.place_and_route(place=hilbert_place)


######################################################################
# Traffic description
######################################################################

# We'll generate bursts of traffic every millisecond
e.burst_period = 1e-3

# We'll choose a particular number (and type) of packet to be
#sent each period
packets_per_period = 32
e.use_payload = True

# We'll run the experiment for a reasonable number of periods,
#allowing some warmup time for the network behaviour to
#stabilise and also adding some cooldown time to ensure all
#vertices have finished recording before stopping traffic
#generation.
e.duration = e.burst_period * 100
e.warmup = e.burst_period * 10
e.cooldown = e.burst_period * 10

# In the experiment we'll generate bursts of traffic with the
#packets being sent in different sized bursts. We'll also repeat
#experiment with and without packet reinjection.
num_steps = 30
for reinject_packets in [False, True]:
    for step in range(num_steps):
        # Work out the proportion of the burst period over which
        #we'll send the packets.
        burst_duty = step / float(num_steps - 1)

        # Work out the time between each packet being sent,
        #we'll use this as the timestep for the traffic
        #generator (which will generate one packet per timestep
        #during the burst).
        timestep = (e.burst_period * burst_duty) /
                    packets_per_period

        # Don't bother trying things with too-tight a timestep
        #since the traffic generator cannot generate packets
        #that fast.
        if timestep < 2e-6:
            continue

        with e.new_group() as g:
            e.reinject_packets = reinject_packets
            e.burst_duty = burst_duty
            e.timestep = timestep

            # We'll add the duty and reinjection option to the
            #results tables
            g.add_label("duty", e.burst_duty)
            g.add_label("reinject_packets", e.reinject_packets)

######################################################################
# Running the experiment
######################################################################

# Record various counter values
e.record_sent = True
e.record_blocked = True
e.record_received = True
e.record_local_multicast = True
e.record_external_multicast = True
e.record_dropped_multicast = True

# Run the experiment
results = e.run(ignore_deadline_errors=True)


######################################################################
# Result plotting
######################################################################

totals = results.totals()

# Scale from 0.0 (nothing received) to 1.0 (every packet which
#was actually sent was received).
totals["received"] /= totals["sent"] * fan_out

# Scale from 0.0 (no packets were sent) to 1.0 (every packet we
#tried to send was sent without being blocked by backpressure).
totals["sent"] /= totals["sent"] + totals["blocked"]

# Scale from 0.0 (no packets dropped) to 1.0 (every MC packet
#routed was dropped).
totals["dropped_multicast"] /= (totals["local_multicast"] +
  totals["external_multicast"])

# Plot with matplotlib
import matplotlib.pyplot as plt

tr = totals[totals["reinject_packets"] == True]
tn = totals[totals["reinject_packets"] == False]

# Plot results with reinjection enabled with solid lines
plt.plot(tr["duty"], tr["sent"], label="sent", color="b")
plt.plot(tr["duty"], tr["received"], label="received", color="g")
plt.plot(tr["duty"], tr["dropped_multicast"],
  label="dropped", color="r")

# Plot results with reinjection disabled with dashed lines
plt.plot(tn["duty"], tn["sent"], linestyle="dashed", color="b")
```

```
# During the experiment we'll ramp up the injection rate and see
#how many packets arrive at their destination.
num_steps = 30
for step in range(num_steps):
    with e.new_group() as g:
        e.probability = step / float(num_steps - 1)
        g.add_label("packets_per_second",
                    ((1.0 / e.timestep) *
                     e.probability *
                     vertices_per_chip *
                     packets_per_timestep))

results = e.run()

totals = results.totals()

# Scale sent-packet count to packets per-second
totals["sent"] /= e.duration

# Plot with matplotlib
import matplotlib.pyplot as plt
plt.plot(totals["packets_per_second"], totals["sent"])
plt.legend()
plt.xlabel("Offered load (Packets per second)")
plt.ylabel("Accepted load (Packets per second)")
plt.show()
```

```
plt.plot(tn["duty"], tn["received"], linestyle="dashed",
  color="g")                                                    plt.legend()
plt.plot(tn["duty"], tn["dropped_multicast"], linestyle="dashed", plt.xlabel("Network duty")
  color="r")                                                    plt.show()
```