Building and Operating Large-Scale SpiNNaker Machines

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING.

2016

Jonathan Heathcote

School of Computer Science

Revision 3ef7fe859288cd086b84d0f2e3384d8b65438d6b Sun, 23 Oct 2016 19:37:27

Contents

Abstract **12**

Lay abstract **13**

Declaration **14**

Copyright **15**

Acknowledgements **16**

1 Introduction **17**

2 The SpiNNaker architecture **23**

2.1 Neural simulation 23

2.2 Alternative neural simulation solutions 25

2.3 Network architecture 26

2.4 The SpiNNaker router 28

2.4.1 Routing tables 29

2.4.2 Timeouts 32

2.5 The hexagonal torus topology 33

2.6 Scaling up SpiNNaker machines 37

2.7 Conclusions 38

3 Building large SpiNNaker machines **41**

3.1 Cabling non-hexagonal torus topologies 42

3.2 Partitioning hexagonal torus topologies 44

3.3 Folding & interleaving hexagonal toruses 46

3.3.1 Rectangularisation 46

3.3.2 Uncrinkling 48

3.3.3 Folding 49

3.3.4 Choosing a transformation 49

3.4 A SpiNNaker case study 53

3.4.1 Mapping into physical cabinets 54

3.4.2 Cable selection and routing 56

3.4.3 Installation practicality 57

3.4.4 Thermal implications 61

3.4.5 Maintenance 61

3.4.6 Verification 62

3.5 Conclusions 63

4 Finding shortest path vectors in SpiNNaker's network **65**

4.1 Shortest path vectors 66

4.1.1 Torus networks 67

4.2 Related work 69

4.2.1 Hexagonal mesh networks 69

4.2.2 Hexagonal torus networks 70

4.3 The Irregular Quadrant (IQ) method 74

4.3.1 Computing shortest path vectors 74

4.3.2 Computing distances 78

4.3.3 Computational efficiency 79

4.4 Generating all shortest path vectors 80

4.5 Conclusions 82

5 Fault tolerant routing for SpiNNaker 83

5.1 Related work 84

5.1.1 Multicast routing in SpiNNaker 85

5.1.2 Fault tolerant routing 87

5.2 Partial Graph Search (PGS) repair 91

5.3 Evaluation and results 93

5.3.1 Traffic patterns 93

5.3.2 Fault models 94

5.3.3 Base routing algorithm 96

5.3.4 Algorithm execution time 96

5.3.5 Routing table usage 98

5.3.6 Network congestion 101

5.4 Conclusions 106

6 Placing applications in large SpiNNaker machines 107

6.1 Placing neural simulations 108

6.2 Related work 110

6.2.1 Application placement algorithms 110

6.2.2 Circuit placement algorithms 118

6.3 Placement by simulated annealing 123

6.3.1 Representation 124

6.3.2 Algorithm description 126

6.4 Evaluation 133

6.4.1 Baseline algorithms 133

6.4.2 Existing applications 134

6.4.3 Scalability 141

6.5 Conclusions 146

7 Conclusions and future research **149**

7.1 Answers to research questions 150

7.2 Future research 153

7.2.1 Warehouse-scale cabling 153

7.2.2 Cabling assistance for other architectures 154

7.2.3 Congestion mitigation 155

7.2.4 Reducing placement execution time 157

7.2.5 Benchmarking 158

7.3 Closing remarks 158

A Partitioning hexagonal toruses **161**

A.1 Tiling 162

A.2 How *not* to tile a hexagonal torus 163

A.3 Triads of triples 164

A.4 Wrapped triples 165

B Minimising hexagonal mesh coordinates **167**

C Software **171**

Bibliography **173**

This thesis contains 30 332 words and 136 figures.

List of Figures

2.1 A Leaky Integrate-and-Fire (LIF) neuron. 24
2.2 SpiNNaker chips connected to their six neighbours. 27
2.3 Block diagram of a SpiNNaker chip. 27
2.4 SpiNNaker's 40 bit and 72 bit multicast packet format. 28
2.5 Multicast routing example. 31
2.6 SpiNNaker router architecture. 32
2.7 A hexagonal torus topology. 34
2.8 Visualisation of a hexagonal torus topology as a torus. 34
2.9 Bisections of torus topologies. 36
2.10 Hexagonal torus vs. H-torus topology. 37
2.11 SpiNNaker boards and their topology. 39

3.1 Folding and interleaving a ring network. 43
3.2 Folding each axis of a hexagonal torus topology. 44
3.3 Partitioning of torus topologies into units. 45
3.4 Rectangularisation of parallelogram partitioned toruses. 47
3.5 Rectangularisation of wrapped triple partitioned toruses. 47
3.6 Patterns of wrap-around connections in sliced systems. 47
3.7 Uncrinkling units into a 2D grid. 50
3.8 Elimination of long wrap-around links by folding. 51
3.9 Physical architecture of a SpiNNaker machine. 54

3.10 Mapping cabling from abstract to physical space. 55
3.11 Cabling plan for a 1200 board SpiNNaker machine. 55
3.12 Cable lengths in a 1200 board SpiNNaker machine. 56
3.13 BlueGene/Q cable installation. 57
3.14 Interactive software guiding cable installation. 58
3.15 Two cabinet SpiNNaker machine cable installation times. 59
3.16 The five cabinet SpiNNaker system. 60

4.1 Shortest path routes in a 2D mesh network. 66
4.2 Finding shortest paths in a 2D torus topology. 67
4.3 Magnitudes of shortest path vectors in a 2D mesh. 68
4.4 An example hexagonal mesh network topology. 69
4.5 Magnitudes of shortest path vectors in a hexagonal torus. 71
4.6 The twelve candidate vectors considered by the INSEE method. 72
4.7 The four quadrants defined by the XYZ-protocol. 74
4.8 The four irregular quadrants defined by the IQ method. 75
4.9 A Python implementation of the IQ Method. 77
4.10 Mean execution times of shortest path vector functions. 79
4.11 Two distinct shortest path vectors in a hexagonal torus. 80
4.12 Distinct shortest path vectors in non-square toruses. 80

5.1 Multicast routes produced by merging unicast routes. 85
5.2 The ESPR and NER multicast routing algorithms. 86
5.3 A deadlock in a ring network. 87
5.4 Deadlock-free routing using DOR in a 2D mesh topology. 88
5.5 Avoiding faults using ‘lamb’ nodes and DOR. 89
5.6 PGS repair example. 92
5.7 An example of a 3-centroid traffic distribution. 94

5.8 The two link fault models considered. 95
5.9 Mean execution time of NER routing and PGS repair. 97
5.10 Routing table sizes for 1000 random networks. 100
5.11 The impact of a HSS link fault. 101
5.12 Distribution of routes-per-chip for 1000 random networks. . . . 103
5.13 Maximum throughput achieved on SpiNNaker. 105

6.1 Partitioning of a 3D volume for a $3 \times 4 \times 2$ torus. 111
6.2 Space-filling curves in 2D mesh and torus topologies. 112
6.3 Relationship between space filling curve offset and 2D distance. 113
6.4 Graph adjacency matrix before and after RCM permutation. . . . 114
6.5 Application placement by recursive partitioning. 117
6.6 Historical core counts for the 'Top500' supercomputers. 119
6.7 An optimal 1D quadratic placement solution. 121
6.8 Selecting vertices to swap. 127
6.9 Net cost estimation functions for hexagonal toruses. 129
6.10 Estimation error of various cost functions. 130
6.11 Application benchmark placement performance. 137
6.12 Packet dropping in Nengo SpiNNaker simulations 140
6.13 Packet dropping in circuit simulator simulations. 142
6.14 Placer scalability with respect to graph size. 144
6.15 Placer scalability with respect to fan out. 145

7.1 Multiple rows of interconnected cabinets. 153
7.2 The irregular edges of a partitioned SpiNNaker system. 156

A.1 Partitioning schemes for 2D and hexagonal torus topologies. . . 162
A.2 Tiling a hexagonal torus with parallelograms. 163
A.3 A parallelogram tiles a hexagonal torus. 163

A.4 A wraped hexagon does not tile a hexagonal torus. 163
A.5 Triples share translational symmetry with flat-topped hexagons. 164
A.6 Triads of triples tile a hexagonal torus. 165
A.7 A wrapped triple with four layers. 165
A.8 Logical connectivity of chips in a triad of SpiNNaker boards. . . 166

B.1 $(1, 1, 1)$ in a hexagonal mesh or torus. 168

List of Tables

3.1 Recommended transformations for folding hexagonal toruses. . 53
3.2 Cable installation times for various SpiNNaker machines. 59

6.1 SpiNNaker application benchmarks. 135

Abstract

BUILDING AND OPERATING LARGE-SCALE SPINNAKER MACHINES

Jonathan Heathcote

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2016.

SpiNNaker is an unconventional supercomputer architecture designed to simulate up to one billion biologically realistic neurons in real-time. To achieve this goal, SpiNNaker employs a novel network architecture which poses a number of practical problems in scaling up from desktop prototypes to machine room filling installations.

SpiNNaker's hexagonal torus network topology has received mostly theoretical treatment in the literature. This thesis tackles some of the challenges encountered when building 'real-world' systems. Firstly, a scheme is devised for physically laying out hexagonal torus topologies in machine rooms which avoids long cables; this is demonstrated on a half-million core SpiNNaker prototype. Secondly, to improve the performance of existing routing algorithms, a more efficient process is proposed for finding (logically) short paths through hexagonal torus topologies. This is complemented by a formula which provides routing algorithms with greater flexibility when finding paths, potentially resulting in a more balanced network utilisation.

The scale of SpiNNaker's network and the models intended for it also present their own challenges. Placement and routing algorithms are developed which assign processes to nodes and generate paths through SpiNNaker's network. These algorithms minimise congestion and tolerate network faults. The proposed placement algorithm is inspired by techniques used in chip design and is shown to enable larger applications to run on SpiNNaker than the previous state-of-the-art. Likewise the routing algorithm developed is able to tolerate network faults, inevitably present in large-scale systems, with little performance overhead.

Lay abstract

BUILDING AND OPERATING LARGE-SCALE SPINNAKER MACHINES

Jonathan Heathcote

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy, 2016.

SpiNNaker is a supercomputer designed to simulate neural networks, such as those that make up the brain. Unlike biological experiments, simulations running on SpiNNaker allow neuroscientists to control and observe the behaviour of these networks while avoiding the need for animal experiments. Though it is extremely unlikely that SpiNNaker will ever 'think', it may lead to a better understanding of the functions and disorders of the brain.

Like most supercomputers, SpiNNaker is made up of many smaller computer processors interconnected by a network. When the largest planned machine is completed, SpiNNaker will contain over one million computer processors – each responsible for simulating several hundred neurons – together capable of simulating neural networks similar in scale to a cat's brain.

This thesis makes three contributions towards the construction and operation of full-sized SpiNNaker machines. Firstly, I devised a new way of organising the physical components which make up a large SpiNNaker machine so that only short cables are required to construct its network making SpiNNaker cheaper and easier to build. Secondly, I developed a method that makes it possible for the processors in SpiNNaker to communicate reliably, even when some connections are faulty – an unavoidable situation in practice. Finally, I adapted a technique normally used to design computer chips to assign neural models to SpiNNaker's processors. This method ensures connected neurons are assigned to nearby processors inside SpiNNaker's network meaning that signals between neurons have to travel less distance and are not as likely to get in each other's way. This makes it possible to simulate larger and more complex neural networks.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.manchester.ac.uk/library/aboutus/regulations`) and in The University's policy on presentation of Theses

Acknowledgements

It is often said that it is not *what* you know but *who* you know. Throughout the course of my PhD I've been exceptionally lucky to have been helped along by a great number of people.

Both my supervisor, Jim Garside, and co-supervisor, Steve Furber, have each spent countless hours patiently discussing and explaining all manner of things to me while giving me tremendous freedom in my project. Jim's office door has always been open to my unexpected interruptions, be it about work, writing or walking. Likewise, Steve has always managed to find time for both technical and frivolous endeavours alike. I'm also hugely grateful to Luis Plana and Steve Temple who have been rich sources of sage advice, insightful questions and have both patiently suffered many of my foolish queries.

Various parts of the work in this thesis (and numerous side projects) would not have been possible if not for a multitude of discussions and collaborations with Javier Navaridas, Simon Davidson and Dave Clark. I'm also indebted to Andrew Mundy and Jamie Knight, both of whom have donated so much time and effort towards discussing, verifying and using software implementations of the ideas in this thesis.

The injection of lunchtime silliness by Andrew and Jamie, along with Andrew Webb, Amanieu d'Antras and other CDT members always brightened my day. Thanks too to the many students and staff of the School of Computer Science who have all helped fill the last few years with fun. Of course, I am also very grateful for the funding the School has provided for my research.

I cannot thank Ann-Marie, my wonderful wife, enough for being by my side. She has given me so much support, love and kindness and endured a lifetime's quota of conversations about hexagons. Finally, the rest of my family deserve enormous gratitude for all their encouragement, especially my parents who are to blame for starting me down this path.

Chapter 1

Introduction

The Spiking Neural Network Architecture (SpiNNaker) is a novel supercomputer architecture designed to simulate biologically realistic models of brains in real-time [FT07]. Though neurons, the building blocks of the brain, are relatively well understood, their complex interactions remain mysterious. Just as understanding the workings of a transistor is insufficient to understand a microprocessor, neuroscientists believe that neurons in isolation cannot explain the brain and that understanding their connectivity is key [Eli13, ET14]. Experiments on real brains, however, are fraught with difficulty. In addition to ethical concerns, variations between individuals can be significant. Furthermore, it is only possible to record tens or hundreds of the brain's trillions of signals at once and even then only limited control is possible over which signals are recorded. Computer simulations of models of neural networks, however, enable researchers to develop repeatable experiments and gain complete visibility of any signal and any neuron. Models such as Spaun [ESC+12], built from millions of simulated neurons, have shown great promise in expanding our understanding of higher-level brain functions such as memory and simple problem solving. Unfortunately, these models are expensive to simulate due to

their enormous implicit parallelism, requiring hours of serial compute time to simulate each second of neural activity. As well as being inconvenient, this precludes the use of robotics to immerse these models in real-world environments and also limits studies of long-term behaviours such as learning.

SpiNNaker is designed to enable the real-time simulation of models containing up to one billion neurons – approximately 1 % of a human brain or ten mouse brains [FTB06]. To achieve this goal, the largest planned SpiNNaker machine will contain over one million low-powered computer processors, interconnected by a bespoke network architecture.

SpiNNaker's large processor count matches the current trend in supercomputers where processor counts are growing exponentially [Meu16], mimicking the growth of the number of components in the processors themselves predicted by Gordon Moore's famous 'law' [Moo75]. As a result of this growth, the interconnection networks which enable these processors to work together have grown in importance [DT01]. Network designers must carefully balance performance against practicality and financial cost. SpiNNaker's network is no exception to this rule and, as the systems scale up from desktop prototypes to machine room scale installations, the reality of building and exploiting these machines presents an array of challenges.

As in all supercomputers, SpiNNaker's network interconnects its processors in a particular network topology which defines how different processors may communicate with each other. Unlike the tree and N-dimensional torus topologies found in contemporary supercomputers [DT04, chapter 3], SpiNNaker employs a 'hexagonal torus topology'. In this topology, nodes fit together in a honeycomb-like pattern where messages may 'hop' from node to node to reach their destination. As we will see in chapter 2, the hexagonal torus topology, in theory, sits at a 'sweet spot' in terms of network performance and practical-

ity. As the first known large-scale installation of a hexagonal torus topology, however, there remain a number of practical challenges for large SpiNNaker machines.

As supercomputer networks have grown in scale to millions of processors, the task of dealing with previously rare faults has grown. Though fault rates in networks remain consistently low, architectures such as SpiNNaker may have hundreds of thousands of links meaning that even fault rates of a fraction of a percent will impact tens or hundreds of links. To enable reliable operation, networks must be able to adapt the routes taken by messages to avoid faulty links and nodes. The techniques conventionally employed are often closely tied to a particular network architecture and because of this SpiNNaker's novel network architecture demands its own approach.

Another challenge introduced by the growing scale of supercomputers is making efficient use of network resources. Communicating processes should be located on logically 'nearby' nodes to reduce network load. The neural models for which SpiNNaker is designed are often described abstractly, rather than geometrically, using modelling languages such as PyNN [DBE+08] and Nengo [BBH+13]. Consequently, communication requirements can be irregular making finding 'good' placements of neurons onto processors non-trivial.

This thesis addresses the practical challenges of scaling up the SpiNNaker architecture in a real-world setting summarised by these research questions:

1. Can the hexagonal torus topology be deployed and used in real, large-scale systems?

2. Does SpiNNaker's router architecture help, or hinder fault tolerance?

3. How can the parts of a neural simulation be placed onto a large hexagonal torus topology to reduce network load?

Chapter 2 introduces the SpiNNaker architecture and, in particular, describes its hexagonal torus topology and network architecture.

In chapter 3, I develop a cabling scheme for hexagonal torus topologies which enables arbitrarily large networks to be constructed using only short, inexpensive cables. This theoretical work is then evaluated through the construction of a range of prototype SpiNNaker systems. The largest of these prototypes contains over half a million processor cores and spans several machine room cabinets. In addition, I propose the use of built-in diagnostic facilities to assist technicians performing network installation and maintenance. This technique was found to greatly reduce the effort required and the number of mistakes made.

In chapters 4 and 5 I develop new routing techniques for SpiNNaker's network. Chapter 4 describes an improved approach to finding the shortest paths through hexagonal torus topologies, an integral part of many routing algorithms. This approach is cheaper to compute than the state-of-the-art and, unlike previous efforts, is able to discover all valid short paths through the topology. This theoretical advance brings hexagonal torus topologies in line with conventional topologies by providing routing algorithms with complete information about the paths available to them. In chapter 5 I propose a fault tolerant routing algorithm for SpiNNaker which is able to avoid arbitrary static fault patterns with minimal performance overhead. A key finding of this chapter is that the flexibility afforded to fault tolerant routing algorithms by SpiNNaker's unconventional router architecture is what facilitates the low overheads reported.

Finally, in chapter 6, I explore the problem of application placement in SpiNNaker's network. As in other networks and applications, neural simulations should be arranged so that communication occurs primarily between

processors close together in the network to control network load. Due to the irregular connectivity and large scale of the neural models expected to run on SpiNNaker, an automated approach is necessary. I develop a novel placement algorithm based on algorithms used for circuit layout in computer chips. My algorithm is found to allow some larger neural models to run on SpiNNaker for the first time while enabling other applications to run at greater speeds. In addition, synthetic benchmarks containing over one million processes indicate that this algorithm should handle the anticipated demands of the neural models expected to run on large-scale SpiNNaker installations.

Chapter 2

The SpiNNaker architecture

SpiNNaker is a massively parallel computer architecture designed to simulate biologically realistic neural models [FT07]. In this chapter we will explore this unconventional architecture in detail, starting with its purpose before focusing on its most distinctive feature: its network.

2.1 Neural simulation

Human brains contain billions of neurons connected together by trillions of 'synapses'. Neurons communicate by transmitting and receiving 'spikes' through their synapses. Each spike is 'valueless' in that a spike's only significant features are when it arrived and where it came from.

Though some detailed models of the electrochemical processes occurring inside neurons are computationally intensive, simplified models such as the Leaky Integrate-and-Fire (LIF) model can be implemented in just a handful of CPU instructions [VG11]. Figure 2.1 illustrates a simple LIF neuron in which incoming spikes cause charge to build up (integrated) which, over time, 'leaks' away. If an incoming spike causes the charge to rise above a certain threshold,

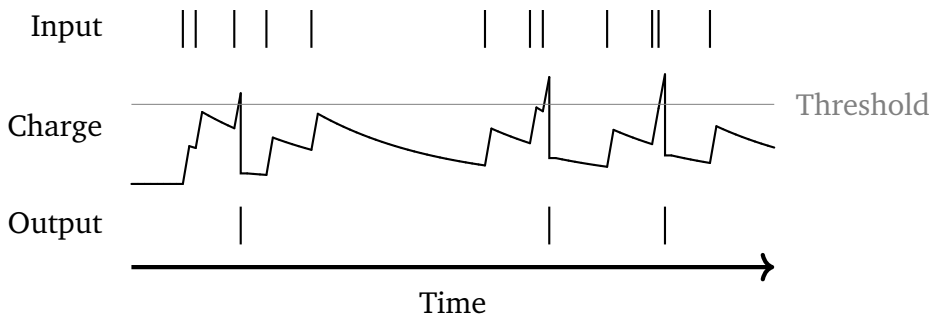


Figure 2.1: A Leaky Integrate-and-Fire (LIF) neuron.

the neuron 'fires' producing an outgoing spike. Despite the simplicity of this model, large neural networks such as Spaun [ESC+12], built entirely from LIF neurons, exhibit complex behaviours such as fine motor control and problem solving.

The computational expense of large-scale simulations of networks of LIF-like neurons does not arise from the cost of modelling the neurons but instead from distributing spikes. In biology, neurons produce spikes at an average rate of 10 Hz and synapses connect each neuron's output to (order) 1000 neurons [NLMA+09]. Consider an example neural model with 7×10^7 neurons, approximately the number in a house mouse and $1/10^{\text{th}}$ of the design target of SpiNNaker. This network might produce 7×10^8 spikes per second. Because each neuron connects to many others, this equates to 7×10^{11} spikes being received per second. If each spike were transmitted as a UDP datagram containing a 32 bit payload, the total network throughput required for this simulation would be 179.2 Tbit s^{-1}. At the time of writing, this is more than double the bisection bandwidth (the theoretical worst-case throughput) of the world's most powerful supercomputer [Don16].

2.2 Alternative neural simulation solutions

Architectures such as IBM's Blue Gene [CCWS11] and Cray's XK7 [Oak16] employ powerful compute nodes interconnected by networks designed to transfer multi-kilobyte blocks of data. Since neural simulations have relatively light computational requirements and communicate only small pieces of data (spikes), this type of architecture is poorly suited to the task. In spite of this, neural simulators such as NEST [GD07], have been developed and used with some success. To reduce the overhead of transmitting spikes in supercomputer networks, NEST groups spikes into batches which are transmitted in a single network packet. This approach is possible because spikes are often modelled as taking some number of milliseconds to reach their destination and so buffering delays can be incorporated into the simulation. Unfortunately, neural models with smaller transmission delays still suffer a loss of efficiency. Further, due to the mismatch of computational and communication needs, neural simulations can be wasteful of supercomputer resources. Knight *et al.* [KTK+16] found that a Cray XC-30 supercomputer uses around 45× more power than SpiNNaker when simulating an identical neural network.

SpiNNaker is not the only alternative to conventional supercomputer architectures aimed at neural simulations. Numerous researchers have created a range of custom silicon solutions for neural simulation [MS10]. These so-called 'neuromorphic' devices implement neuron models directly in silicon. The BrainScaleS [SBG+10] and Neurogrid [BGM+14] projects, for example, have built a neural simulators composed of large arrays of analogue neuron models with programmable interconnect. Analogue neuron models can be very energy efficient but suffer from environmental sensitivity and calibration difficulties making them difficult to use in practice. Fully digital neural simulators like Bluehive [MFM+12] avoid these problems at the expense of some energy

efficiency and can be much easier to use.

Though hardware implementations of neural models can be fast and energy efficient, these architectures become restricted to simulating only specific classes of neuron models making them unsuitable for many researchers. SpiNNaker, by contrast, uses software to implement its neural models allowing it to cater to a much wider range of models.

2.3 Network architecture

SpiNNaker's architectural target is to support real-time simulations of models containing up to one billion neurons. Since neural models such as LIF are inexpensive to compute and many neurons can be simulated independently in parallel, SpiNNaker employs many small, energy efficient general-purpose processors [FT07]. To support the unusual communication requirements of neural simulations, a bespoke interconnection network is used which is the background to this thesis.

The fundamental building block of the SpiNNaker architecture is the SpiNNaker chip (figures 2.2 and 2.3) [FLP+13]. Each chip contains eighteen low power ARM968 processor cores each capable of simulating between 200 and 2000 LIF neurons in real-time [MKSF15]. Each core has a total of 96 kB of private Tightly-Coupled Memory (TCM) and shares access to 128 MB of on-chip SDRAM with other cores on the same chip. Finally, each chip contains a programmable router which routes network packets to and from the local cores and six neighbouring SpiNNaker chips. SpiNNaker machines are constructed by combining many SpiNNaker chips.

Processor cores can communicate by sending and receiving network packets forwarded by routers through the network. Since SpiNNaker's network is

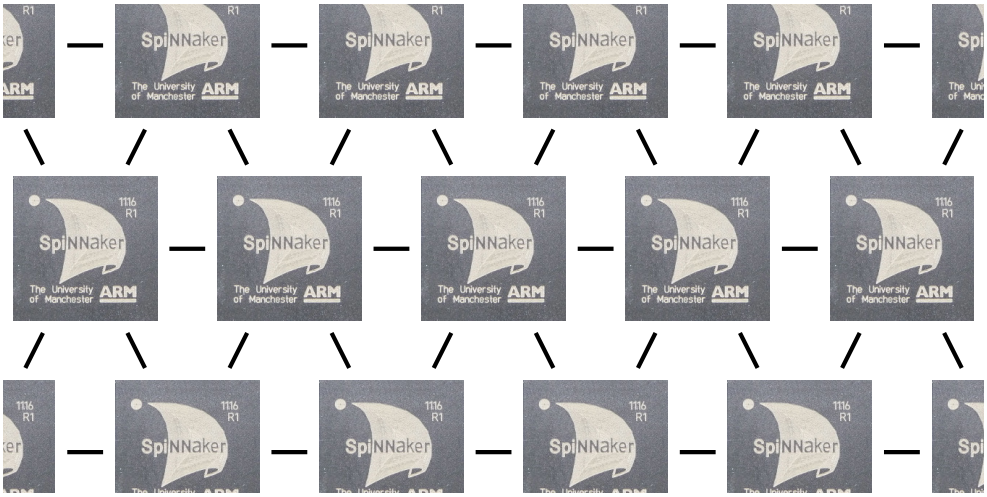


Figure 2.2: SpiNNaker chips (actual size) connected to their six neighbours.

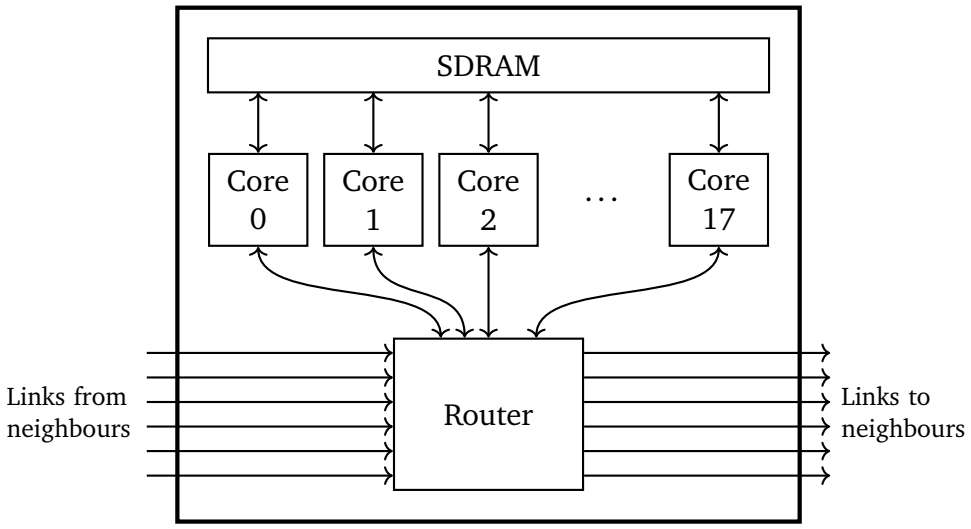


Figure 2.3: Block diagram of a SpiNNaker chip.

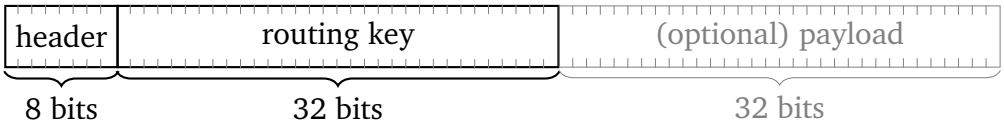


Figure 2.4: SpiNNaker's 40 bit and 72 bit multicast packet format.

designed to transmit neural spike events efficiently, individual network packets are small, either 40 bits or 72 bits compared with tens or hundreds of bytes in typical network architectures.

Since at biological timescales a computer network delivers packets 'instantaneously', the time at which a spike is produced is implicitly indicated by the time it is received in a real-time simulation.

Consequently, the only information which must be explicitly encoded is the identity of the neuron which produced the spike. In SpiNNaker, a spike may be represented by a single 40 or 72 bit 'multicast packet' whose format is illustrated in figure 2.4. The 8 bit header is used by SpiNNaker's routers to determine the type of packet and the 32 bit 'routing key' is used to identify the neuron which produced the packet. The routing key is also used by SpiNNaker's routers to determine how the packet should be directed through the network. The optional 32 bit payload is not used by conventional spiking neural simulations [GRDF10] but has been exploited to enable more efficient simulation of a particular class of neural models [MKSF15].

2.4 The SpiNNaker router

The SpiNNaker router employs an unconventional design which, despite its compact size and small energy requirements, implements a flexible multicast routing mechanism. Unlike some routers which employ hard-coded routing rules [DT04, chapter 8], the SpiNNaker router uses a programmable 'routing

table' to determine how packets should be forwarded. In addition, to avoid deadlocks, SpiNNaker's router employs a simple, timeout-based mechanism which exploits the ability of neural networks to tolerate occasional missing packets. As we will see in chapter 5, this mechanism greatly simplifies the task of routing in SpiNNaker's network. In this section we will look at SpiNNaker's router in greater detail.

2.4.1 Routing tables

When a multicast packet arrives at a SpiNNaker router (either from a local core or a neighbouring chip), the router looks up the routing key in its routing table. This table consists of 1024 programmable table entries, each specifying a routing key and mask to match and a set of routes. If a multicast packet's key is matched by a routing entry, the packet is forwarded along every route specified by that entry, potentially duplicating the packet. This 'multicast' technique allows packets to be transmitted once but received in a number of places while making efficient use of the network [NLP+12].

Though routing table entries are in finite supply (1024 entries per router), it is still possible for many thousands of traffic flows to be routed through a single router. The key and mask in each routing entry matches against the 32 bits of a routing key as either '1', '0' or 'X' ('don't care'). This means that a single routing entry may, for example, be used to match all routing keys with a certain prefix. If a routing key is not matched by any entry in the routing table then the packet is 'default routed' in a straight line. For example if a packet with an unmatched key is received from the chip to the left, the packet will be default routed to the chip on the right. By assigning routing keys and defining routes carefully to take advantage of these features, the number of routing entries required by a simulation is greatly reduced [DNGF12].

Consider the simplified example in figure 2.5 in which a number of (4 bit) routing table entries have been configured in the routers of a small SpiNNaker network. If a packet with the routing key `1011` is transmitted by a core in the chip labelled $(0,0,0)$, this will match the first routing table entry on that chip and will be routed to chip $(1,0,0)$. Here the packet once again matches the first routing entry and is routed to chip $(2,1,0)$. No entries match on this chip so the packet is default routed to $(3,2,0)$. On this chip, the packet matches a routing entry which routes the packet to core 7. In this example, default routing allows only three routing table entries to direct a packet through four chips.

As a second example, if a packet with the routing key `0010` is transmitted by a core on chip $(0,0,0)$, this key will be matched by the second routing entry since `X`s in the table entry will match both `1`s and `0`s in the corresponding bits of the routing key. When the packet arrives at chip $(1,1,0)$ the matching routing entry forwards the packet to both $(1,2,0)$ and $(2,1,0)$ simultaneously. The copy of the packet arriving at $(1,2,0)$ is routed to core 5 on that chip. Meanwhile, the copy forwarded to $(2,1,0)$ is duplicated again with one copy being routed to core 11 and another being routed to chip $(3,2,0)$. Here the packet is finally delivered to core 6. In this example, the ability of the router to multicast (duplicate) packets as they pass through the network meant that sending one copy of the packet was sufficient to reach three destination cores. In addition, by using `X`s in the routing table entry, the same routing entries are sufficient to route packets with the keys `0000`, `0001`, `0010` and `0011`.

In spite of these mechanisms, it is still possible for an application to run out of routing table entries. As we will see in chapter 6, by arranging applications appropriately within SpiNNaker's network, routing table usage can be reduced. In addition, other behaviours of SpiNNaker's router may be exploited to com-

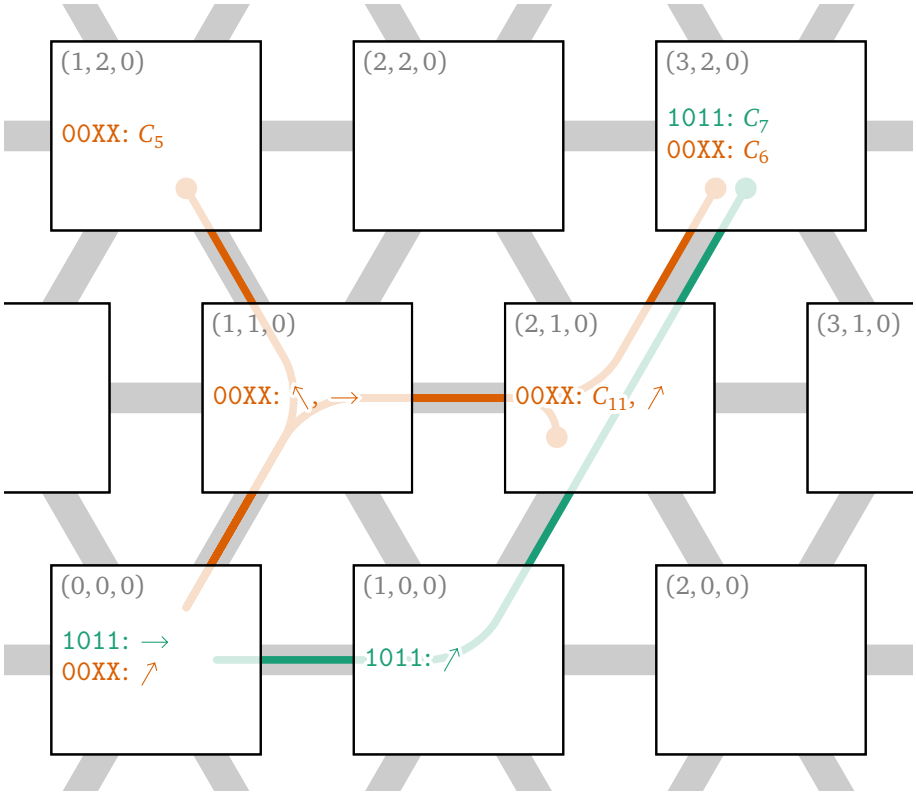


Figure 2.5: Multicast routing example with 4 bit routing keys. Each box represents a SpiNNaker chip whose router has been programmed with the routing entries shown. Grey lines mark connections between chips. Coloured lines illustrate example routes.

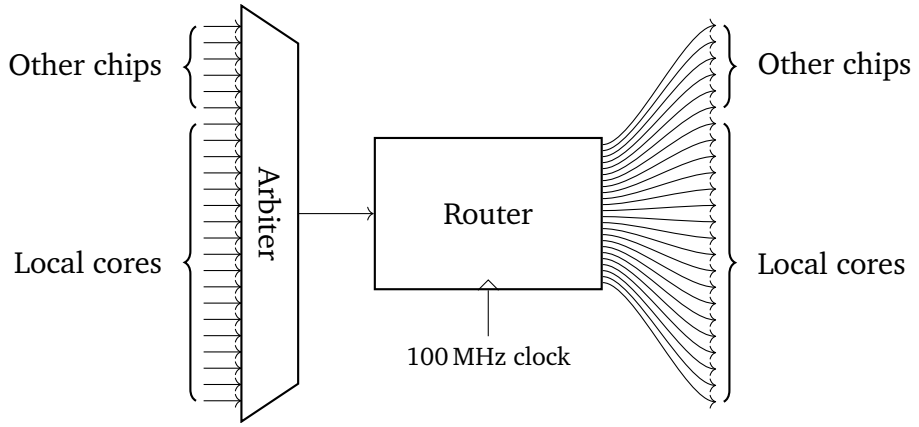


Figure 2.6: SpiNNaker router architecture.

press an application's routing tables further [MHG16], however discussion of the specific techniques employed are beyond the scope of this thesis.

2.4.2 Timeouts

SpiNNaker's router is built on a pipeline architecture. As shown in figure 2.6, the router is fed packets by an arbiter which serialises packets arriving from other chips and local cores. Every (100 MHz) clock cycle, the router pipeline accepts one packet from the arbiter and routes a packet to one or more output links. If any of the required output ports are busy then the packet is not forwarded to any output link and the pipeline stalls. Once a packet has been blocked for a programmable timeout, it is 'dropped' (discarded) and routing continues as usual for the next packet in the pipeline. Links become blocked while transmitting packets or waiting for the remote receiver to become ready. For example, a receiving processor core may be busy performing some computation or a receiving router may be blocked waiting for some of its outputs to become ready.

The timeout-based packet dropping mechanism is designed to defuse dead-

locks in the network. For example, if two routers are trying to send each other a packet at the same time they may become deadlocked, each waiting for the other router to accept a packet before continuing. SpiNNaker's timeout mechanism breaks deadlocks by dropping packets which have been blocked for some time and therefore may be in a deadlock. Once a packet has been dropped it is left to software to either tolerate the missing packet or trigger a retransmission. In neural simulations, as in biology, the loss of a single spike is unlikely to have a significant impact on the behaviour of a neural model and therefore these simulations are inherently tolerant of occasional dropped packets. During application loading and other system tasks a higher-level, software-driven protocol based on acknowledgements and retransmissions is used to ensure guaranteed delivery.

2.5 The hexagonal torus topology

Each SpiNNaker chip is a node in a 'hexagonal torus topology' as illustrated in figure 2.7. Network packets sent by SpiNNaker's processor cores may 'hop' through several nodes in the network to reach their intended destination. In each hop, a packet may advance one node along one of the three axes of the topology. For example, a packet sent by the node labelled α (in the bottom-left corner) to the node labelled β, might take a route with the following sequence of hops: X^+, X^+, Z^-. Packets sent from α to γ might take the route: X^-, X^-, Y^+, Y^+. The first hop of this route 'wraps around' from the bottom-left node to the bottom-right node in a single hop.

The wrap-around connections in the topology are what give it the 'torus' part of its name. Laid out as a sheet (figure 2.8a) the topology may be rolled up such that the top and bottom nodes become adjacent, forming a tube (fig-

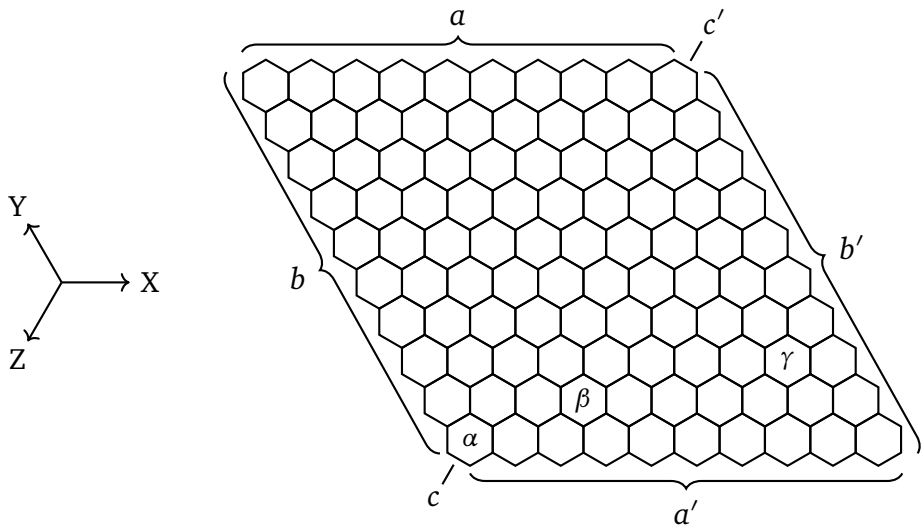


Figure 2.7: A hexagonal torus topology. Each hexagon represents a node (i.e. a SpiNNaker chip). Touching nodes are directly connected. Nodes on edges a, b and c are also directly connected to the corresponding nodes on edges a', b' and c', respectively. The three axes of the hexagonal torus topology, 'X', 'Y' and 'Z' are also shown.

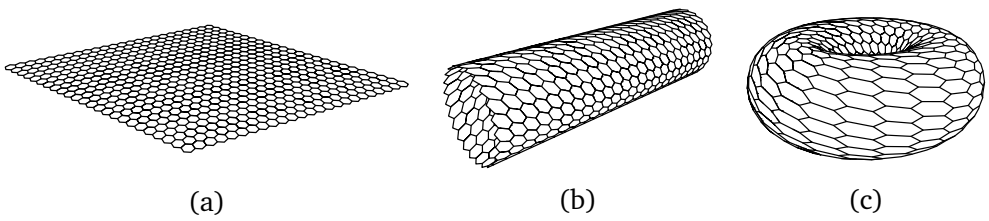


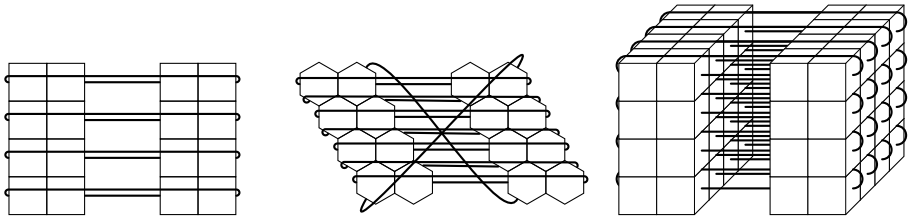
Figure 2.8: Visualisation of a hexagonal torus topology as a torus.

ure 2.8b). This tube can then be bent to bring together the nodes at the ends of the tube to form a torus as shown in figure 2.8c.

A hexagonal torus topology is typically defined in terms of its width and height along the X and Y axes respectively. For example, the hexagonal torus in figure 2.7 is 10×10. The nodes in a hexagonal torus topology are addressed using hexagonal coordinates of the form (x, y, z) [Ami15]. The bottom left node (labelled α in the figure) has the coordinate $(0, 0, 0)$ and other nodes are assigned coordinates according to the number of hops along each dimension from $(0, 0, 0)$, for example node β has the coordinate $(2, 0, -1)$. Because the hexagonal torus topology's axes are non-orthogonal, it is possible to define several coordinates for the same location. For example $(3, 1, 0)$ and $(1, -1, -2)$ are also valid coordinates for node β. These dual coordinates emerge from the fact that adding $(1, 1, 1)$ to a coordinate produces an equivalent, but different, coordinate. The cause of this particular phenomenon is explained in detail in appendix B and related phenomena will be discussed in chapter 4.

The hexagonal torus topology was chosen over a more conventional network topology – such as a 2D or 3D torus (sometimes known as a 2-ary N-cube or 3-ary N-cube respectively) [DT04, chapters 3 and 5] – due to its balance of theoretical performance and practicality. The bisection bandwidth of a topology indicates the theoretical worst-case total throughput the network is able to sustain [DT04, chapter 1]. In networks with homogeneous link throughput, bisection bandwidth is determined by the number of links cut by a balanced bisection of the network. Figure 2.9 illustrates bisections of several torus topologies.

In an $N \times N$ 2D torus topology, the bisection bandwidth is $2N$ links and each node requires four links. The hexagonal torus topology requires six links per node but provides double bisection bandwidth ($4N$ links). The 3D torus topol-



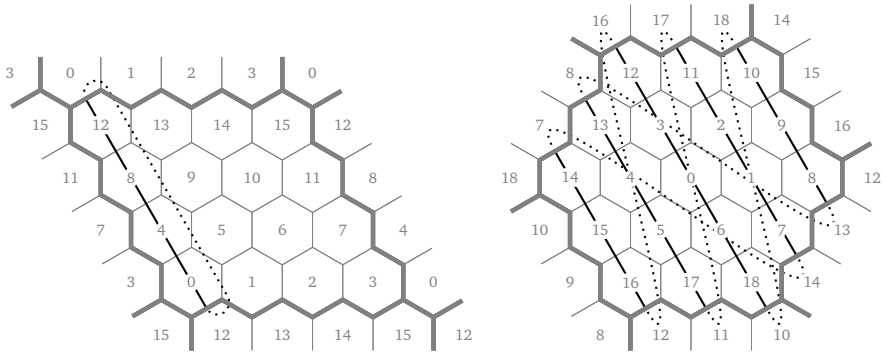
(a) 2D Torus (b) Hexagonal Torus (c) 3D Torus

Figure 2.9: Bisections of torus topologies. Connections cut by the bisection are drawn as lines.

ogy also requires six links per node but by connecting the nodes differently achieves a bisection bandwidth of $8N$ links. The 3D torus topology, however, comes at a price – when immersed into the (approximately) 2D space provided by a large machine room or row of server cabinets, some connections require long cables. By contrast, the 2D and hexagonal torus topologies are both inherently two dimensional and consequently do not suffer from this effect. The hexagonal torus topology, therefore, shares the practicality of construction of a 2D torus while still gaining some of the performance of a 3D torus topology. In addition, because nodes in a hexagonal torus topology have a greater number of links, greater redundancy is available in the network to tolerate faults.

Most torus topologies, including hexagonal, 2D and 3D toruses, have a related 'mesh' topology. These mesh topologies maintain the same general connectivity structure as their toroidal counterparts but omit the wrap-around links. In practice, this saves a small number of links at the expense of halving the network's bisection bandwidth. Because of their poorer performance, mesh networks are rarely used as the basis of a network architecture. Mesh networks, however, are occasionally formed when a torus is partitioned into several smaller sub-networks to allow multiple users to share a system [Hea16b].

The hexagonal torus topology is not to be confused with the 'H-torus' topology. The H-torus also uses a hexagonal tiling of nodes and even wraps this



(a) Hexagonal torus (b) H-torus

Figure 2.10: Hexagonal torus vs. H-torus topology. Each numbered hexagon represents a node. The thick outline indicates the bounds of the topology after which the network repeats. In each topology, the path taken by advancing in the Y^+ direction from the node labelled '0' is shown.

tiling into a torus-like topology [ZYW08]. However, H-toruses have very different characteristics to the hexagonal torus topology and bear similarities to the 'twisted torus' topology [CMV+10]. Figure 2.10 illustrates one difference in the way paths wrap around the peripheries of both topologies.

2.6 Scaling up SpiNNaker machines

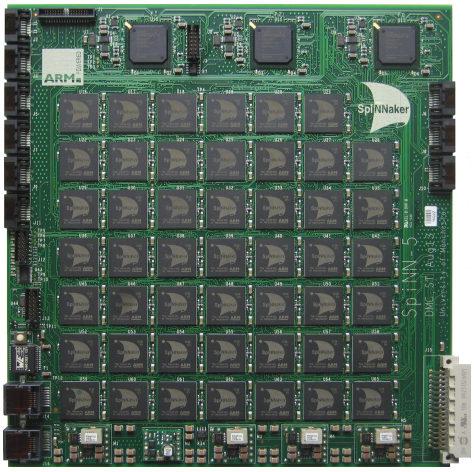
To build large SpiNNaker systems comprising tens of thousands of SpiNNaker chips, groups of forty-eight chips are mounted onto circuit boards as illustrated in figure 2.11a. These boards may be connected together to form larger systems. Figure 2.11b shows a prototype three board system. Though the chips are physically arranged in a (nearly) 7×7 grid on each SpiNNaker board, they logically form a hexagonal 'wrapped triple' [Dav13] (see appendix A) which fit together as illustrated in figure 2.11c. The labelled exposed corners of the three forty-eight chip boards connect together to form a 12×12 hexagonal torus topology as illustrated in figure 2.11d. Larger SpiNNaker machines are

assembled by combining more boards.

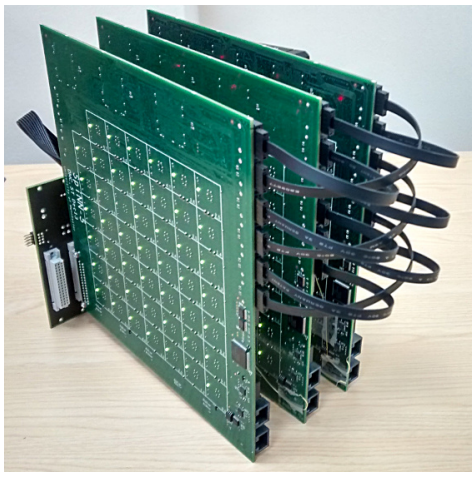
SpiNNaker chips on the same circuit board connect using low power links requiring sixteen wires each. If this link technology were used to connect chips on neighbouring boards, each pair of boards would need to be connected with a 128 wire cable. Cables and connectors supporting this many signals are expensive, less reliable and physically large. Instead, chip-to-chip connections between boards are multiplexed and demultiplexed onto a single High-Speed Serial (HSS) link [AC05] carried via commodity S-ATA cables which are often used to connect hard disks in desktop computers and servers [Ser08]. The six high-speed links are implemented by three onboard FPGAs (the three large chips at the top of the SpiNNaker board) and are logically transparent to the underlying network. The underlying technology and the choice of S-ATA cables limits each board-to-board connection to spanning, at most, one metre gaps. In chapter 3 I present a cabling scheme for hexagonal torus topologies which enable large SpiNNaker systems to be assembled using only short cables between boards.

2.7 Conclusions

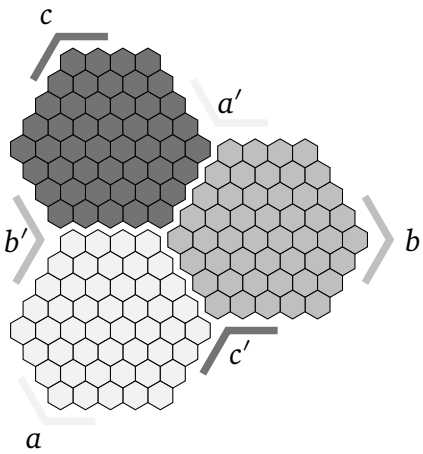
The SpiNNaker architecture has been designed to enable the simulation of large biologically realistic neural models in real-time. To support this, its network architecture takes on an unconventional design based on a custom router and hexagonal torus topology. In the remainder of this thesis, I will tackle a number of the challenges in scaling up the SpiNNaker architecture outlined in this chapter.



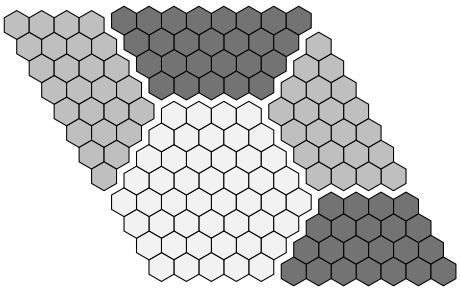
(a) A SpiNNaker board



(b) Three board prototype



(c) Three board topology



(d) . . . as a parallelogram

Figure 2.11: SpiNNaker boards and their topology.

Chapter 3

Building large SpiNNaker machines

Like any supercomputer, physically assembling a large SpiNNaker machine poses many practical challenges in terms of arranging, installing and maintaining the hundreds of metres of network cables required. For conventional architectures and network topologies, techniques are well understood and embodied by industry standards such as TAI-942 [Ass06]. SpiNNaker's use of the hexagonal torus topology renders existing approaches insufficient.

In the first part of this chapter I extend existing techniques for mapping network topologies into standard data centre physical infrastructure to support the hexagonal torus topology. This mapping is designed to ensure that all cables are kept short (under 1 m) to reduce costs and simplify the network hardware required. The techniques described introduce little overhead in cable length over existing torus wiring schemes and confirm the suitability of the hexagonal torus topology for real-world applications.

The second part of this chapter uses SpiNNaker as a case study on the suitability of the mappings introduced in this chapter. In this case study I con-

sider various SpiNNaker systems ranging in size from desktop machines to multi-cabinet machine room installations. As well as validating the cabling schemes introduced in this chapter I also describe a new technique which improves the efficiency of the cable installation process. As a consequence of SpiNNaker's fine-grained connectivity, the cabling is unusually dense, exacerbating the complexity of the cabling patterns to be installed. By exploiting network diagnostics hardware and on-board LEDs to guide cable installation, construction of large SpiNNaker machines takes a matter of hours rather than the days reported for other architectures. In addition, preliminary experiments suggest that neither the maintainability nor cooling performance of the system are hampered by the dense cabling employed.

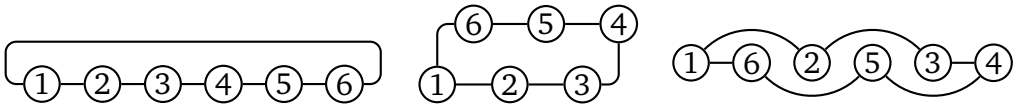
In this chapter, the term *unit* refers to the smallest physical component between which network interconnection cables are installed. For example, in a SpiNNaker machine a unit is a 48-chip board while in a data centre network a unit might be a server blade.

3.1 Cabling non-hexagonal torus topologies

Naïve arrangements of torus topologies, hexagonal or otherwise, feature physically long 'wrap-around' connections which connect units at the peripheries of the system. Long connections can be problematic for several reasons:

Performance: Signal quality diminishes as cables get longer, requiring the use of slower signalling speeds, increased error correction overhead or more complex hardware.

Energy: Some energy is lost in cables; longer cables lose more signal energy requiring higher drive strengths and/or buffering to maintain signal integrity.



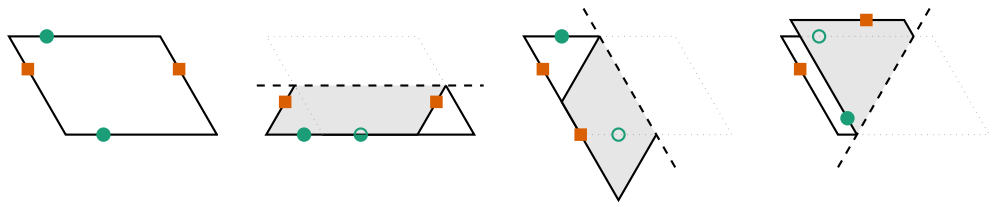
(a) A ring network (b) Folded (c) Folded and interleaved

Figure 3.1: Folding and interleaving a ring network to reduce maximum cable length.

Cost: Shorter cables are cheaper than long ones. Longer cables imply more cabling in a given space making the task of cable installation and system maintenance more difficult, increasing labour costs by as much as 5× [CCE+12].

In some cases, long connections in supercomputers may be eliminated by creative physical organisation of the system. For example, the distinctive 'C'-shaped design of early Cray supercomputers was chosen to reduce the lengths of physical connections and thus improve system performance [THO13]. Unfortunately, this approach does not scale up in the general case and requires potentially expensive bespoke physical infrastructure. Instead the need for long cables is often eliminated by folding and interleaving units of the network [DT04, chapter 5]. This process is illustrated for a 1D torus topology (a ring network) in figure 3.1. A naïve arrangement of units in this topology results in a long cable connecting the units at the ends of the ring (figure 3.1a). To eliminate these long connections, half of the units are 'folded' on top of the others (figure 3.1b) and then this arrangement of units is interleaved (figure 3.1c). This ordering of units requires no long cables while still observing the physical constraint that units must be laid out in a line.

The folding and interleaving process may be extended to N-dimensional torus topologies by folding each axis in turn. Since all axes are orthogonal in non-hexagonal topologies, the folding process only moves units along the axis being folded. Due to the non-orthogonality of the three axes of the hexagonal



(a) Not folded (b) X (c) Y (d) Z

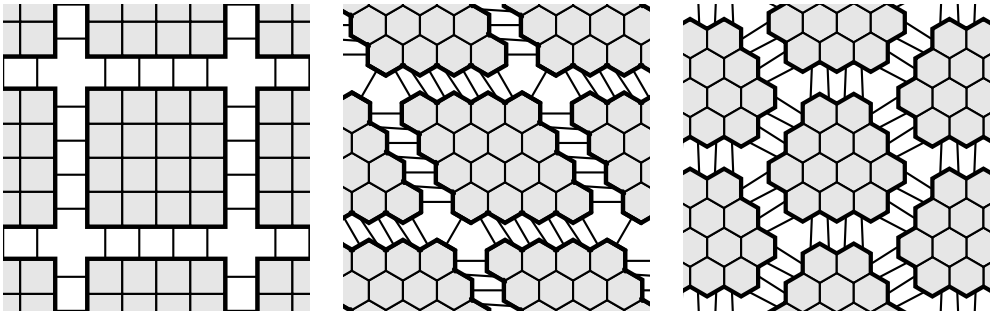
Figure 3.2: Schematics showing folding along each axis of a hexagonal torus topology failing to eliminate wrap-around connections. Same-shaped-and-coloured dots show the endpoints of two example wrap-around connections.

torus topology, this type of folding does not work. As figure 3.2 illustrates, folding along any axis results in connected units on opposing edges not being brought together. For example, when folding along the X axis, the two units marked with a green circle are moved closer together on the Y axis but remain apart on the X axis.

3.2 Partitioning hexagonal torus topologies

The nodes in supercomputer networks are usually relatively small, for example a single chip. Tens of nodes are packed together into a single unit, such as a circuit board or server blade, to simplify assembly and share common power and cooling resources [Gil14, AIHS12]. In commercial supercomputers built on non-hexagonal torus topologies, each unit's nodes represent a hypercube partition of the overall topology as illustrated in figure 3.3a [CEH+11, AIHS12].

An analogous 'parallelogram' partitioning scheme exists for hexagonal torus topologies, however, this results in imbalanced connectivity requirements between neighbouring partitions. In figure 3.3b, for example, the partitions above, below, left and right of the central partition are connected by seven node-to-node connections each while the partitions above-right and below-left are connected by just a single connection each. To simplify assembly, connec-



(a) 2D hypercube (b) Parallelogram (c) Wrapped triple

Figure 3.3: Partitioning of non-hexagonal (a) and hexagonal (b and c) torus topologies into units.

tions between all nodes in a pair of neighbouring partitions are often made by a single cable. If connectivity requirements are imbalanced, as in this example, this may mean multiple connector types may be required, increasing design complexity.

To avoid connectivity imbalance, SpiNNaker uses a 'wrapped triple' partitioning scheme [Dav13], as illustrated in figure 3.3c and explained in detail in appendix A. In this partitioning scheme, the same number of connections connect all six neighbouring units. As explained in the appendix, a hexagonal torus topology is constructed from groups of three partitions.

For completeness, both parallelogram and wrapped triple partitioning are considered in this chapter even though SpiNNaker uses wrapped triple partitioning. The parallelogram partitioning scheme may be more appropriate for architectures where connections between nodes in neighbouring partitions do not share a single connector. In addition, in architectures where a unit corresponds to a single node, this can be treated as a 1×1 parallelogram partition. This special case occurs in coarse-grained architectures and Networks on Chip (NoCs) where nodes are not grouped together into multi-node units.

3.3 Folding & interleaving hexagonal toruses

To exploit the folding technique used by non-hexagonal topologies, the units in a hexagonal torus topology must be mapped into a space with orthogonal coordinates. The choice of transformation to an orthogonal coordinate system can have an impact on how physically far apart logically neighbouring units are in the final arrangement. A good mapping should attempt to reduce 'distortion' which moves adjacent units apart in the final folded and interleaved arrangement.

In this section I propose two transformations which map hexagonal arrangements of units into a 2D orthogonal coordinate space. The first transformation, 'shearing', is general purpose but introduces some distortion. The second transformation, 'slicing', is less general but can introduce less distortion than shearing and therefore may lead to shorter cable lengths.

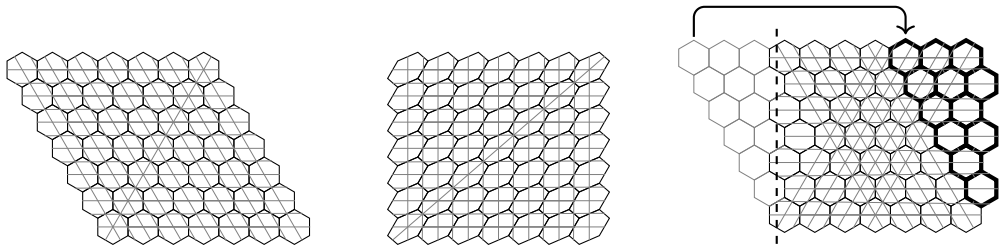
Both the slicing and shearing transformations are carried out in two steps:

Rectangularisation Units are transformed from being laid out in a parallelogram into a rectangular arrangement. The specific transformation used is the key difference between the slicing and shearing transformations.

Uncrinkling Units are mapped into a 2D coordinate system without gaps between units.

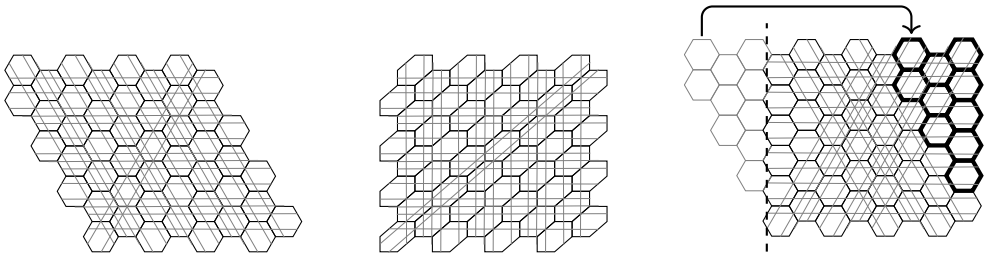
3.3.1 Rectangularisation

The hexagonal torus topology is illustrated in figures 3.4a and 3.5a for parallelogram-partitioned units and wrapped triple units respectively. The first step in the folding process is to rearrange units so that they form a rectangle using one of two techniques: shearing or slicing.



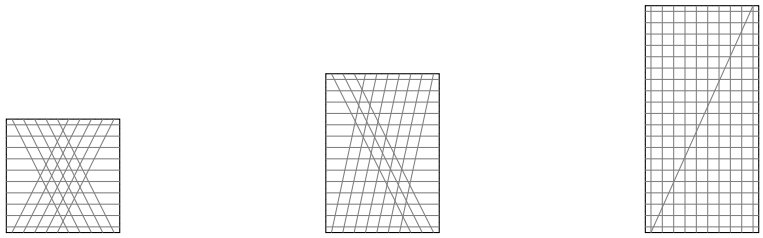
(a) Original (b) Sheared (c) Sliced

Figure 3.4: Rectangularisation of parallelogram partitioned hexagonal toruses. Thin lines show wrap-around links. Pointy-topped hexagons represent parallelogram partitioned units.



(a) Original (b) Sheared (c) Sliced

Figure 3.5: Rectangularisation of wrapped triple partitioned hexagonal toruses. Thin lines show wrap-around links. Flat-topped hexagons represent wrapped triple partitioned units.



(a) 5×5 (b) 5×7 (c) 5×10

Figure 3.6: Schematics showing the patterns of wrap-around connections in sliced systems of various aspect ratios.

The shearing technique applies a 30° shear transformation to distort the arrangement of units so that the X and Y axes of the hexagonal torus topology become orthogonal. This transformation leads to a rectangular arrangement of units as illustrated in figures 3.4b and 3.5b.

The shear transformation introduces some distortion causing connections between units in the Z axis to become $\sqrt{2}\times$ longer. The transformation does not alter the pattern of wrap-around connections: long connections between units on the extreme left and right sides and the top and bottom remain, along with a single connection between the bottom left and top right units.

The slice transformation aims to avoid the elongation of the Z axis by moving the units without distorting their layout. Units protruding from the left-hand-side of the parallelogram are 'sliced off' and moved into the matching gap on the opposite side as illustrated in figures 3.4c and 3.5c. This transformation does not introduce any distortion but changes the pattern of wrap-around connections. Connections from left to right remain while the connections between the top and bottom units now criss-cross (figure 3.6). The proportion of connections going from bottom-left to top-right and from bottom-right to top-left varies depending on the aspect ratio of the topology. Only certain patterns of wrap-around links can be eliminated by folding and, as we shall see later, this limits us in which network topologies can be rectangularised by slicing.

3.3.2 Uncrinkling

Before folding can occur, the rectangularised arrangements of units produced in the previous step must be mapped into a 2D grid. Applied to parallelogram partitions, the shear transformation results in a mapping into a 2D grid with no further distortion (figure 3.7a). For other combinations of transformation and partitioning scheme, the units do not exactly fit a 2D grid. Instead, the

units form ‘crinkled’ rows or columns which may be ‘uncrinkled’ (straightened out) to fit a regular 2D grid as illustrated in figures 3.7b – 3.7d.

3.3.3 Folding

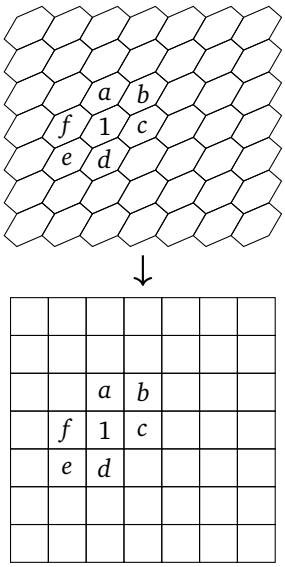
Once a regular 2D grid of units has been formed, this may be folded in the conventional way as illustrated in figure 3.8a. Folding once along each axis eliminates long connections crossing from left to right, top to bottom and from the bottom-left corner to the top-right corner. Any shear-transformed network may be folded this way since its wrap-around connections always follow this pattern. Slice-transformed networks may only be folded like this when their aspect ratio is 1 : 2 when the pattern of wrap-around links is the same as a shear-transformed network.

When ‘square’ networks (i.e. those with a 1 : 1 aspect ratio) are sliced, the network must be folded *twice* along the Y axis as in figure 3.8b to eliminate the criss-crossing wrap-around links. It is not possible to eliminate wrap-around links from sliced networks with other aspect ratios by folding.

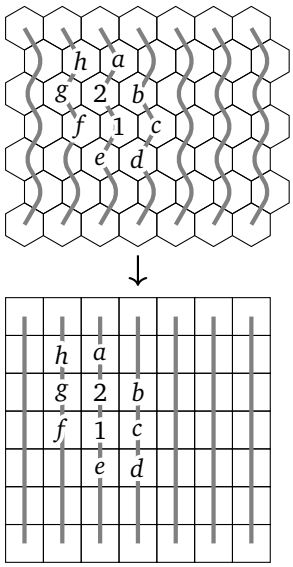
After folding, the units are interleaved, yielding a 2D arrangement of units in which no connection spans the width or height of the system. The maximum connection distance is constant for any network allowing the topology to scale up.

3.3.4 Choosing a transformation

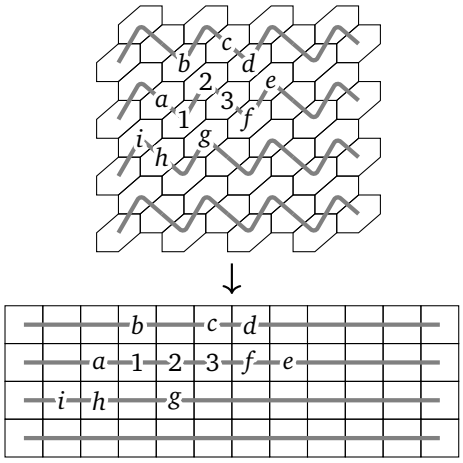
In each step of the transformation from hexagonal torus to a folded and interleaved 2D grid, the distances between connected units may increase. When designing a system, the transformation with the least distortion should be used to minimise the average length of the cables required.



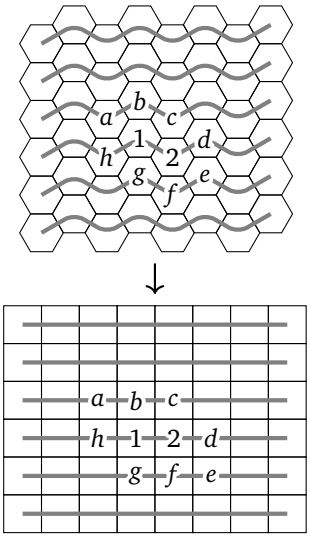
(a) Parallelogram units, sheared



(b) Parallelogram units, sliced

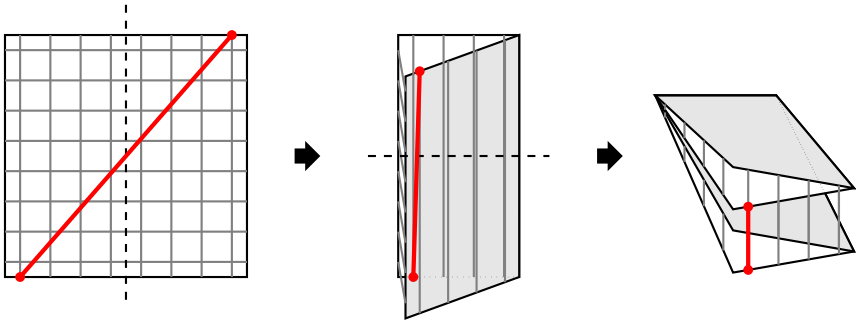


(c) Wrapped triple units, sheared

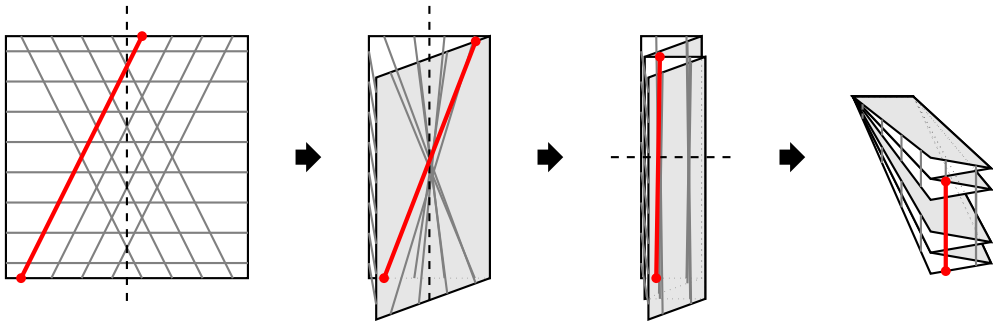


(d) Wrapped triple units, sliced

Figure 3.7: Uncrinkling rectangularised arrangements of units into a 2D grid. Thick lines show how crinkled rows and columns of units are uncrinkled. Annotations show how the relative positions of units change after uncrinkling.



(a) Sheared systems and 1 : 2 sliced systems



(b) 1 : 1 sliced systems

Figure 3.8: Schematic illustrating elimination of long wrap-around links during folding. In each example a single link has been highlighted to aid in following the process.

By referring to figure 3.7 again, it is possible to calculate the overhead introduced by each type of transformation. For example, to compute the overhead introduced by the slicing transformation when applied to units composed of wrapped triples we consider figure 3.7d. The uncrinkling pattern used to transform this topology is a repeating pattern of two units, a pair of which have been labelled 1 and 2 respectively. Unit 1 is immediately surrounded by the six units labelled a, b, c, 2, g and h. Unit 2 is surrounded by the units 1, c, d, e, f and g. Before the transformation, the distance between units is 1; after the transformation is applied this is not always the case. Folding and interleaving into f_x columns and f_y rows also introduces overhead. For each pair of previously neighbouring units in the example, their distances after folding may be computed as follows:

$$
\begin{aligned}
D_{1\leftrightarrow a} &= \sqrt{f_x^2 + f_y^2} & D_{2\leftrightarrow 1} &= f_x \\
D_{1\leftrightarrow b} &= f_y & D_{2\leftrightarrow c} &= f_y \\
D_{1\leftrightarrow c} &= \sqrt{f_x^2 + f_y^2} & D_{2\leftrightarrow d} &= f_x \\
D_{1\leftrightarrow 2} &= f_x & D_{2\leftrightarrow e} &= \sqrt{f_x^2 + f_y^2} \\
D_{1\leftrightarrow g} &= f_y & D_{2\leftrightarrow f} &= f_y \\
D_{1\leftrightarrow h} &= f_x & D_{2\leftrightarrow g} &= \sqrt{f_x^2 + f_y^2}
\end{aligned}
$$

From these values, along with similar formulae for other partitioning schemes and transformations, mean and maximum connection distances may be calculated. The expressions for each combination are as follows:

	1 : 2	Other
Parallelogram	**Either** $D_{\text{mean}} \approx 2.28$ $D_{\text{max}} \approx 2.83$	**Shear** $D_{\text{mean}} \approx 2.28$ $D_{\text{max}} \approx 2.83$
Wrapped triples	**Slice** $D_{\text{mean}} \approx 2.28$ $D_{\text{max}} \approx 2.83$	**Shear** $D_{\text{mean}} \approx 3.00$ $D_{\text{max}} \approx 4.47$

Table 3.1: Recommended transformations for folding hexagonal toruses.

$$D_{\text{mean}} = \begin{cases} \frac{7f_x + 3\sqrt{f_x^2+f_y^2} + \sqrt{(2f_x)^2+f_y^2}}{9} & \text{if sheared wrapped triple units} \\ \frac{f_x + f_y + \sqrt{f_x^2+f_y^2}}{3} & \text{otherwise} \end{cases}$$

$$D_{\text{max}} = \begin{cases} \sqrt{(2f_x)^2 + f_y^2} & \text{if sheared wrapped triple units} \\ \sqrt{f_x^2 + f_y^2} & \text{otherwise} \end{cases}$$

Using these formulae it is possible to determine which approach – shearing or slicing – results in the lowest mean and maximum cable lengths and thus which technique should be used. This is summarised in table 3.1.

3.4 A SpiNNaker case study

As the only known large-scale hexagonal torus-based architecture, SpiNNaker is a good case study for the techniques described in this chapter. Each unit in a SpiNNaker machine is a 48-chip SpiNNaker board forming a wrapped triple partition. Systems of various sizes have been constructed using the techniques introduced in this chapter ranging from twenty-four board 'portable' systems to a five cabinet, half-million core installation with plans in place to build a machine of twice this size in the future.

In this section I describe how the folded and interleaved arrangement of

Figure 3.9: Physical architecture of a SpiNNaker machine.

units produced by the techniques in the previous chapter may be translated into physical arrangements of SpiNNaker boards in a machine room. I then describe how the thousands of S-ATA cables are installed and report on the maintainability and cooling impact of this cabling scheme in practice.

3.4.1 Mapping into physical cabinets

In SpiNNaker systems, the physical architecture used is illustrated in figure 3.9. SpiNNaker boards are installed into card frames containing twenty four boards each. Five frames are mounted into standard, 600 mm wide 19″ cabinets with further cabinets being added, arranged in a row, to scale the system up. The 2D grid of units produced by the folding process described in this chapter is mapped to cabinets and frames as illustrated in figure 3.10.

Figure 3.11 shows the cabling plan for the largest planned SpiNNaker machine. This system will fill ten 19″ cabinets and implement a 240×240 hexagonal torus topology partitioned between 1200 48 chip SpiNNaker boards. The largest gap to be spanned by any cable is 66 cm, well within the 1 m limit on SpiNNaker's interconnect technology.

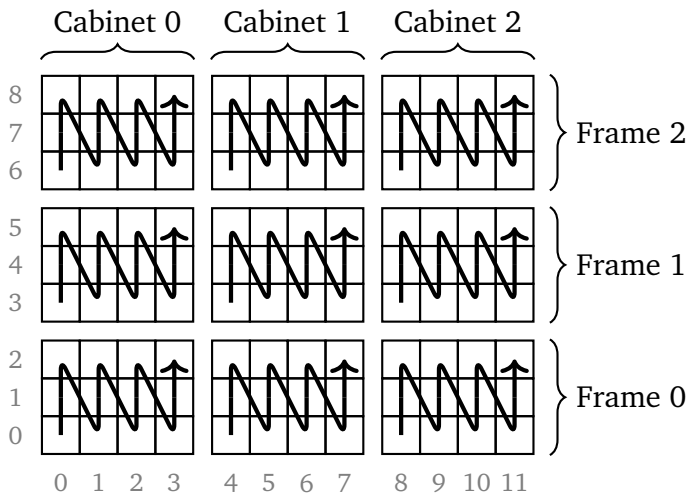


Figure 3.10: Mapping from the abstract folded and interleaved 2D grid layout into physical cabinet and frame positions. Arrows indicate the order in which units (boards) are mapped into each frame, from left to right.

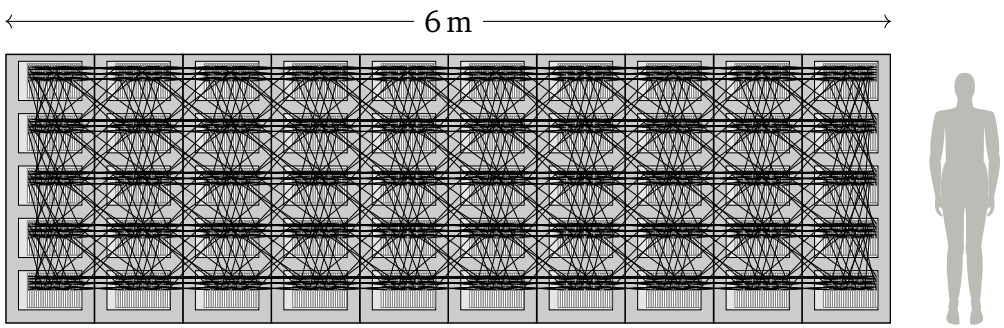


Figure 3.11: Cabling plan for a 1200 board (1 036 800 core) SpiNNaker machine.

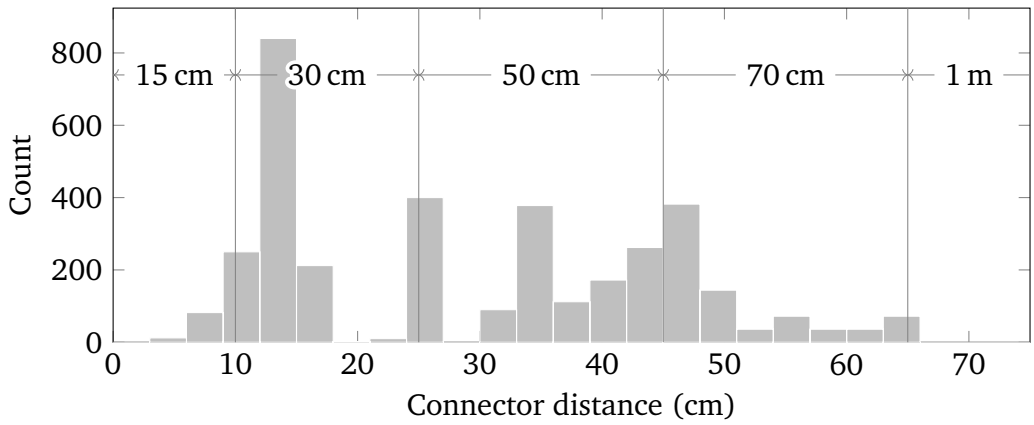
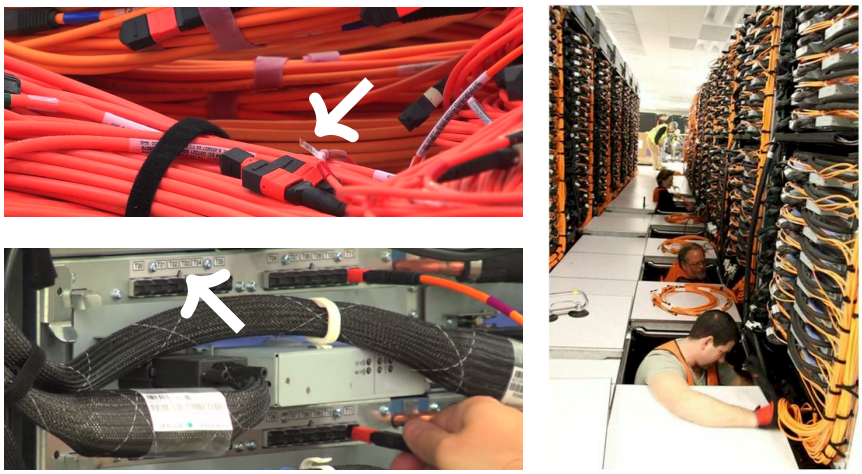


Figure 3.12: Histogram of connection distances in a 1200 board SpiNNaker machine annotated with the selected cable lengths.

3.4.2 Cable selection and routing

Because of the dense packing of SpiNNaker boards, cables span short distances as shown in figure 3.12. Conventional cable management techniques (e.g. cable trays) are not practical. To ensure the reliability and maintainability of SpiNNaker's wiring, cable slack must be carefully controlled. If cables are too tight cables, connectors and SpiNNaker boards can become damaged. When cables are too slack, the excess obstructs access to the machine and can easily become tangled [Cis07].

In this case study the 'rule of (three-)thumbs' proposed by Mazaris [Maz97] is used which suggests that a minimum of 5 cm of slack be provided. As SpiNNaker uses off-the-shelf S-ATA cables, only standard lengths of cable are available. For any given span, the shortest length of cable providing at least 5 cm of slack is used.



(a) Pre-labelled cables and sockets (b) Installation of cables

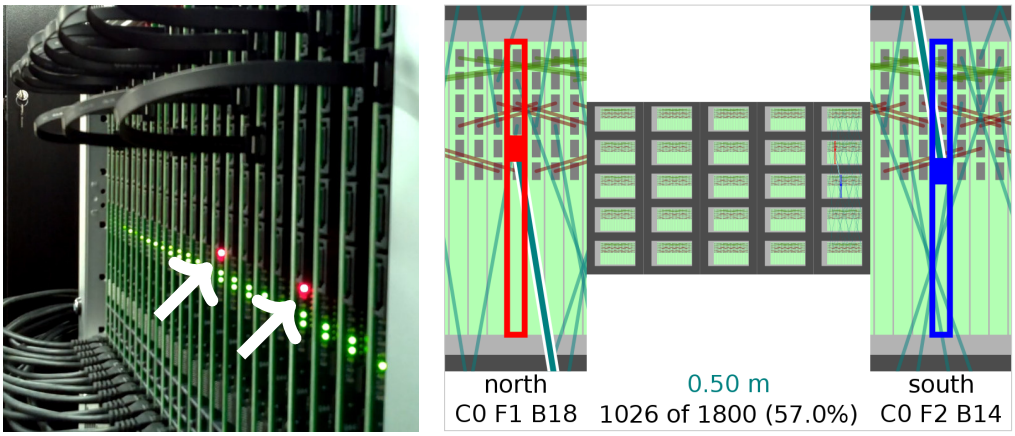
Figure 3.13: BlueGene/Q cable installation [CSC13].

3.4.3 Installation practicality

In other large-scale architectures, the task of cable installation is completed by a team of technicians aided by the use of standardised labelling for cables and sockets as illustrated in figure 3.13 [Ass06]. In these architectures the cabling patterns required are relatively straightforward, thanks to the coarseness of the units used [LMMS07] or they use network topologies whose cabling centres around high-fan-in switches [Cis07, Cse15].

It has been reported in the literature that when copper cables are used, labour costs dominate [PRI+10] and while cable costs are expected to decline, labour costs are not [MYM11]. Many researchers have attempted to control cable installation costs by trying to reduce the number or length of cables required by developing alternative network topologies [CCE+12, PRI+10, MYM11]. Unfortunately, these techniques do not apply to SpiNNaker since its network topology is fixed.

Supercomputer architectures such as BlueGene/Q make use of large custom 'midplane' PCBs in place of cables to connect units within a cabinet [ML13].



(a) Diagnostic LEDs indicate the endpoints of each cable.

(b) A GUI and text-to-speech indicate what type of cable to install.

Figure 3.14: Interactive software guiding cable installation.

This scheme can reduce wiring complexity since only coarser-grain, cabinet-to-cabinet connectivity is implemented by cables. Unfortunately, this technique is expensive and constrains the dimensions of the network topology supported by the machine. Since SpiNNaker is designed to scale from desktop machines to machine room installations, this scheme is not practical.

Due to the high density of units in a SpiNNaker system, the detailed cabling patterns used can be complex, despite their overall regularity. To cope with this complexity, I developed a software system which employs diagnostic hardware built into SpiNNaker, to guide technicians through the cable installation process. As shown in figure 3.14, diagnostic LEDs on each SpiNNaker board are used to indicate which boards to connect. The software also provides step-by-step cabling instructions via a Graphical User Interface (GUI) and audible instructions delivered via headphones. These instructions explicitly specify the length of cable to use for each connection avoiding the common problem of technicians over-estimating the cable length required [Maz97]. Diagnostic registers in the network hardware are then used to verify the correct installa-

Size	Cables	Time	Notes
24 boards	72	10 min	
1 cabinet	360	4 h	Real-time validation not used.
2 cabinets	720	2 h	
5 cabinets	1800	4 h 20 min	Three people working simultaneously.

Table 3.2: Cable installation times for various sizes of SpiNNaker machine.

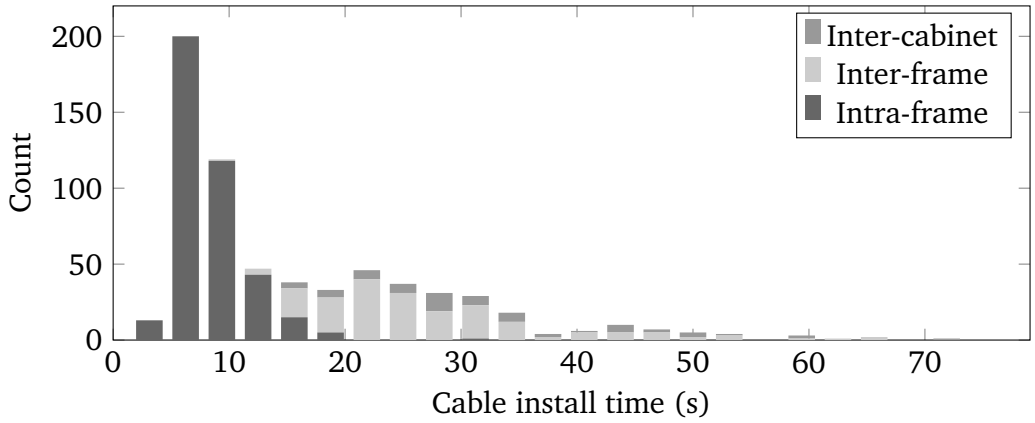


Figure 3.15: Two cabinet SpiNNaker machine cable installation times.

tion of each cable in real-time ensuring that mistakes are highlighted and fixed immediately.

Table 3.2 shows cable installation times for various sizes of SpiNNaker system. The times reported do not include breaks and, with the exception of the five cabinet system, are for the one person working alone. Figure 3.15 shows the histogram of cable installation times for a two cabinet machine. These results confirm the observation by Mudigonda *et al.* that cables which span cabinets and frames take longer to install [MYM11], even though these distances are still very short in SpiNNaker. Compared with commercial installation efforts, per-cable installation times are much shorter for SpiNNaker taking seconds compared with minutes in other architectures [MYM11].

The positive impact of real-time validation of installed cables can clearly

Figure 3.16: The five cabinet SpiNNaker system. The cabinet on the right contains conventional host servers which control SpiNNaker.

be seen by comparing the installation times of the one and two cabinet systems. Though double the size, the two cabinet machine was built in half the time required to build the single cabinet machine. While building the smaller machine, real-time cable validation had not yet been implemented and the installation process was interrupted for several minutes every time a misplaced connection was discovered.

In the three-person cable installation effort employed for the five cabinet system, the guidance software was configured to assign each technician cables in non-overlapping parts of the machine ensuring minimal interference between the technicians. As expected, this renders the problem embarrassingly parallel, as in commercial computer installations.

The completed five cabinet system is photographed in figure 3.16. A time-lapse video showing the construction and cable installation of this machine is also available on YouTube [Hea16a].

3.4.4 Thermal implications

In large SpiNNaker machines, each 24 board frame contains a fan tray which pulls cool air from the front of the frame, between the SpiNNaker boards. The warmed air is then ejected out of the rear of the frame where ducting directs it into industrial chillers. Conventional guidance on data centre design suggests that routing cables in the path of the system's airflow can have a serious impact on cooling performance [Cis07]. To determine what effect the cabling described in this chapter had on SpiNNaker's cooling, a test program was executed to simulate heavy load before and after cable installation. The temperatures reported by the sensors on the top of each SpiNNaker board were sampled at regular intervals and once the overall system temperature stabilised, the mean temperature was recorded.

Before the cabling was installed, the temperature stabilised at 49 °C while after installation it stabilised at 42 °C.

These two data points suggest that the system's temperature is unlikely to have been been seriously impacted by cable installation. Since the two experiments were run on different days (with potentially different ambient temperatures) and are based on a single experiment, it is not possible to infer much more from this result.

3.4.5 Maintenance

At the time of writing, the five cabinet machine is still being commissioned and so the long term maintenance impact of the system's cabling is not known. One important factor in the maintainability of the system is the ease with which faulty boards can be replaced. During commissioning a number of boards have been replaced due to discovery of defects not found during manufacture. These

replacements were carried out by someone not involved in the machine's installation who reported that boards near the centre of a frame (i.e. those most likely to be blocked by unrelated cables) take around ten minutes to replace, including time spent removing and replacing cables obstructing the board being exchanged. By comparison boards at the edge of the machine take around six minutes to replace. Though similar timing reports are unavailable for other architectures, these times appear reasonable in practice and suggest maintenance is not impaired by the wiring plan used.

3.4.6 Verification

The real-time cabling verification process ensures that each board is connected correctly to its neighbours according to the cabling pattern described in this chapter. To verify that this pattern actually implements the desired hexagonal torus topology, SpiNNaker's system software was upgraded to check the correctness of the network topology during boot.

When the system is first booted a single chip is assigned the coordinate $(0, 0, 0)$. From this, its six neighbouring chips may in turn discover their own locations in the topology, followed by their neighbours and so on. Eventually every chip knows its location. If the cabling scheme does not form a valid hexagonal torus topology, some chips will be assigned conflicting coordinates, causing a boot error.

All of the SpiNNaker machines constructed were found to have the correct network topology. Furthermore, the machines constructed have been used successfully in practice, including in later work described in chapters 5 and 6.

3.5 Conclusions

In this chapter I presented a practical method of constructing real-world installations of large hexagonal torus topologies such that long cables spanning the width and height of the system are not required. Two transformations, shearing and slicing, are presented which allow conventional network folding techniques to be applied to hexagonal toruses to eliminate long 'wrap-around' links. Though both techniques incur some overhead in terms of mean and maximum cable length, the maximum cable length does not grow with the size of the network. This result makes hexagonal torus topologies a practical and scalable choice for future systems.

The theoretical results presented in this chapter have been confirmed through the successful construction of several large-scale SpiNNaker machines which implement hexagonal torus topologies. During construction, diagnostic features of SpiNNaker's hardware were employed to guide technicians performing cable installation. This technique was found to be highly effective with cable installation times measured in seconds rather than minutes as reported for other architectures. Surprisingly there was no evidence found of this technique being applied to other architectures and consequently this secondary result may be of interest for future research.

Chapter 4

Finding shortest path vectors in SpiNNaker's network

In the previous chapter we explored the practical challenges of building machines which use hexagonal torus topologies working at the scale of units containing several nodes. To exploit these machines, however, we must also be able to route packets efficiently through the nodes in the resulting network. This chapter tackles the problem of finding shortest path vectors in hexagonal torus topologies. Shortest path vectors are used by many routing algorithms as the basis for route generation. In non-hexagonal torus topologies, finding shortest path vectors is trivial and intuitive but in hexagonal toruses, this is not the case. In this chapter I introduce the Irregular Quadrant (IQ) method, a new technique for computing shortest path vectors in hexagonal torus topologies. This method is cheaper to compute and more general than pre-existing techniques, functioning correctly on hexagonal torus topologies of any aspect ratio.

In some hexagonal torus topologies, many shortest path vectors may exist between a given pair of points. I propose a technique for discovering all possi-

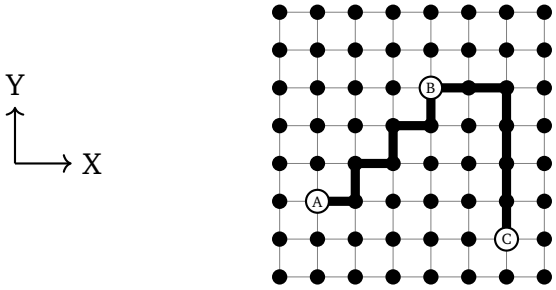


Figure 4.1: An example 2D mesh network with example shortest-path routes from 'A' to 'B' and 'B' to 'C'.

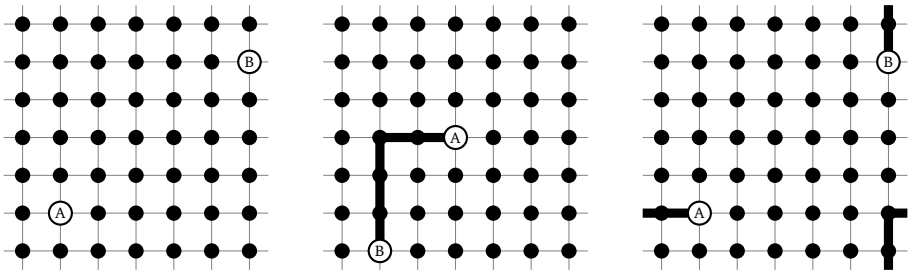
ble shortest path vectors. Using these alternative shortest path vectors, routing algorithms may be able to produce routes which load a network more evenly.

In this chapter, I assume an idealised hexagonal torus topology without faults or other irregularities. The challenge of handling these artefacts of real-world systems will be tackled in chapter 5.

4.1 Shortest path vectors

Many popular routing algorithms for torus topologies, including all published algorithms designed for SpiNNaker [DNGF12, NLP+15], compute a shortest path vector between the endpoints of a route and use this to generate a path through the network. A shortest path vector between two nodes is a vector, $\mathbf{v} = (v_1, v_2, v_3, \ldots)$, whose magnitude, $\|\mathbf{v}\| = |v_1| + |v_2| + |v_3| + \cdots$, is minimal with respect to all possible vectors between those nodes.

In a non-hexagonal mesh topology, shortest path vectors are computed by taking the element-wise difference between the source and destination nodes' coordinates. For example, figure 4.1 shows a 2D mesh topology with three nodes labelled 'A', 'B' and 'C' with position vectors $(1,2)$, $(4,5)$ and $(6,1)$ respectively. The shortest path vector from node 'A' to 'B' is $(4,5)-(1,2) = (3,3)$ and from 'B' to 'C' is $(6,1)-(4,5) = (2,-4)$. A route may be produced by ad-



(a) Original (b) Routed & translated (c) Routed original

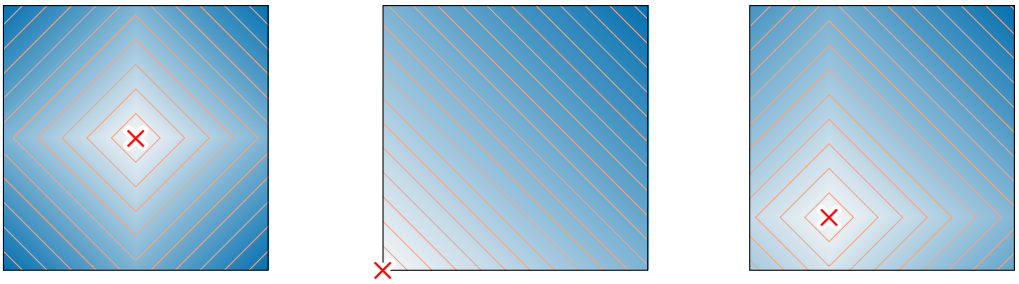
Figure 4.2: Finding shortest paths in a 2D torus topology.

vancing the number of hops specified for each dimension in the shortest path vector. For example, a route from 'A' to 'B' may be constructed from any permutation of the hops $X^+ X^+ X^+ Y^+ Y^+ Y^+$, an example of which is included in the figure. Likewise routes from 'B' to 'C' may be constructed from permutations of the hops $X^+ X^+ Y^- Y^- Y^- Y^-$. Regardless of the order of the hops, the length of the route, $\mathbf{v}$, is given by the magnitude of the shortest path vector, $\|\mathbf{v}\|$.

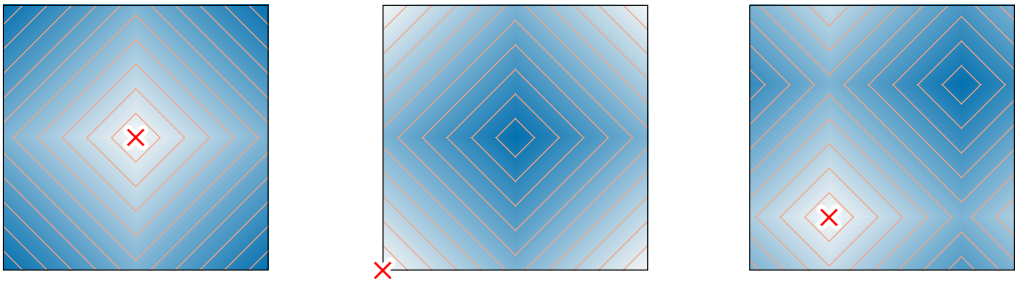
Many popular routing algorithms such as dimension order routing, right-turn only routing and longest dimension first routing [DNGF12] are simply defined as rules for ordering the hops specified by a shortest path vector.

4.1.1 Torus networks

Computing shortest path vectors in non-hexagonal torus topologies is also straightforward. For example, to find the shortest path vector from node 'A' to 'B' in the 2D torus topology shown in figure 4.2a both nodes are translated such that the source node, 'A', is at the centre of the network. The shortest path vector is then computed in the same way as a mesh network (figure 4.2b). Note that, as in this example, translation may cause the destination node to 'wrap around' the network. As illustrated in figure 4.2c, the computed shortest path vector is also valid for the two points prior to translation.



(a) 2D mesh topology



(b) 2D torus topology

Figure 4.3: Plots showing the magnitude of shortest path vectors in a 2D (non-hexagonal) topology from locations marked ×. Darker areas are further away. Contour lines show equidistant points.

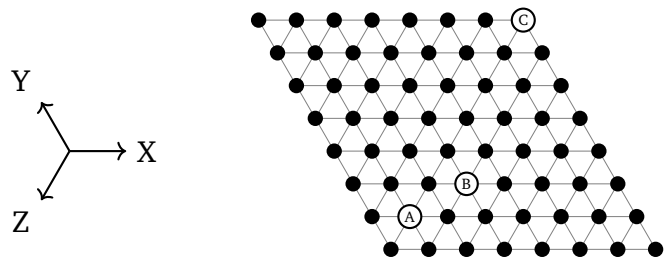


Figure 4.4: An example hexagonal mesh network topology.

This procedure works because vectors from the centre of a non-hexagonal torus topology to any other point are identical to those in a corresponding mesh topology. For example, in figures 4.3a and 4.3b we can see that the magnitude of the shortest path vectors from the centre of a mesh and torus grow identically. Conversely, the magnitudes of vectors from other locations in mesh and torus topologies do not match.

4.2 Related work

The problem of finding shortest path vectors in hexagonal mesh topologies has been widely considered and formulations may be found in a variety of applications, including computer games [Ami15]. Hexagonal toruses, by contrast, have only received limited attention. In this section I briefly summarise the solutions proposed for hexagonal mesh topologies before more deeply examining existing solutions for hexagonal torus topologies.

4.2.1 Hexagonal mesh networks

In hexagonal mesh topologies it is conventional to define three 'axes' X, Y and Z as shown in figure 4.4 [Ami15]. In this example, the three labelled nodes 'A', 'B' and 'C' may be given position vectors such as $(1, 1, 0)$, $(3, 2, 0)$ and

$(0,0,-7)$ respectively. As in other mesh networks, a vector between two nodes is found by subtracting the nodes' vectors. For example, a vector from 'A' to 'B' is $(3,2,0)-(1,1,0)=(2,1,0)$. This vector can then be converted into a route in the same way as a mesh network by taking any permutation of the three hops $X^+ X^+ Y^+$.

As explained in detail in appendix B, there are a multitude of vectors between any two points in a hexagonal mesh. For example, the vectors $(1,0,-1)$ and $(3,2,1)$ also reach node 'B' from 'A'. However, for a given pair of nodes, there is always a single, unique vector whose magnitude is minimal which is given by the function:

$$\text{minimiseVector}(x,y,z) = (x,y,z) - \text{median}(x,y,z) \cdot (1,1,1)$$

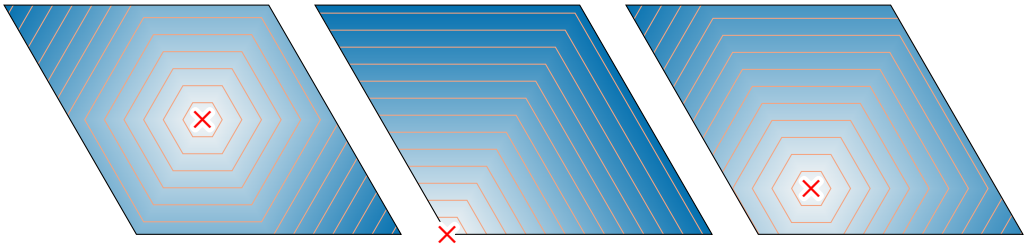
For example, the vector $(3,2,1)$ from 'A' to 'B' is minimised as follows:

$$\begin{aligned}\text{minimiseVector}(3,2,1) &= (3,2,1) - \text{median}(3,2,1) \cdot (1,1,1) \\ &= (3,2,1) - (2,2,2) \\ &= (1,0,-1)\end{aligned}$$

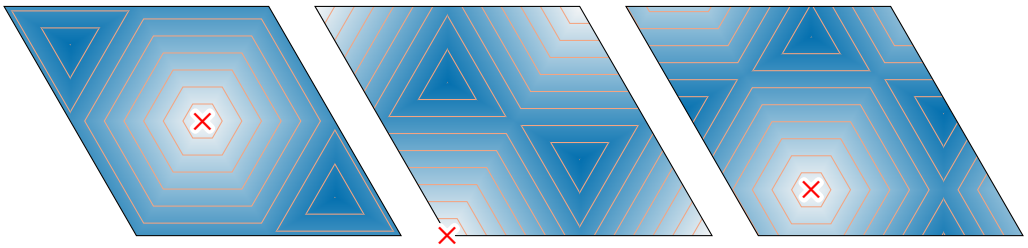
A side-effect of this is that a minimised vector will always contain at least one zero element, meaning that shortest path routes will use at most two of the three available dimensions.

4.2.2 Hexagonal torus networks

Unfortunately, the translation technique used for non-hexagonal toruses cannot be used in a hexagonal torus. As illustrated in figures 4.5a and 4.5b, shortest path vectors from the centre, or any other part of a hexagonal mesh net-



(a) Hexagonal mesh topology



(b) Hexagonal torus topology

Figure 4.5: Plots showing the magnitude of shortest path vectors in a hexagonal torus topology from locations marked ×. Darker areas are further away. Contour lines show equidistant points.

work, do not grow in magnitude in the same way that those of a hexagonal torus network do. I am aware of two pre-existing approaches to computing shortest path vectors in hexagonal toruses. These are described below.

INSEE Method

The INSEE interconnect simulator has been used in all published research into SpiNNaker's hexagonal torus interconnect to date [NLMA+09, GHN+15]. Internally INSEE finds shortest path vectors by selecting the shortest of a set of twelve candidate vectors known to always contain a shortest path vector.

The twelve vectors considered are illustrated in figure 4.6 and are constructed as follows. First a shortest path vector from the source to target node is constructed as if the network was a 2D mesh producing a vector (Δ_X, Δ_Y). A second 2D vector, (Δ'_X, Δ'_Y), is also defined:

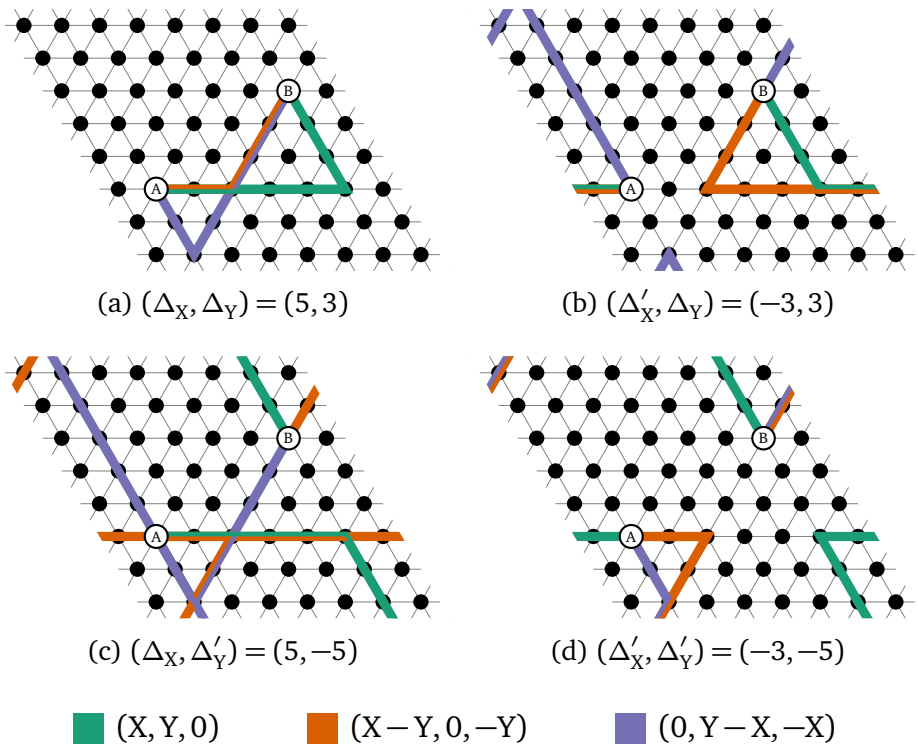


Figure 4.6: The twelve candidate shortest-path vectors considered by the INSEE method represented as dimension-order routes. $W = H = 8$, $(\Delta_X, \Delta_Y) = (5, 3)$ and $(\Delta'_X, \Delta'_Y) = (-3, -5)$.

$$\Delta'_X = \Delta_X - \text{sign}(\Delta_X)W$$
$$\Delta'_Y = \Delta_Y - \text{sign}(\Delta_Y)H$$

Where W and H are the width and height of the network respectively. This vector describes a route from the source to destination node that *always* wraps around the peripheries of both the 'X' and 'Y' dimensions.

Two further 2D vectors, (Δ'_X, Δ_Y) and (Δ_X, Δ'_Y) may be defined which wrap around just the X or Y axis, respectively.

Each of the four 2D vectors may be converted into three hexagonal 3D vectors in which one element of the vector is zero. In total this results in twelve different vectors which cover all combinations of wrapping and non-wrapping routes and all combinations of axes used. The vector with the smallest magnitude must be the shortest path vector.

This method can find shortest path vectors in hexagonal torus topologies of any aspect ratio but, compared with the XYZ-protocol (described next), is relatively clumsy and slow to compute.

XYZ-Protocol

Hoffmann and Désérable described the XYZ-protocol for computing shortest path vectors in hexagonal toruses with equal width and height [HD15, HD11]. First, the source and destination nodes are translated such that the source node lies at the centre of the topology. The authors observe that from the centre of the topology, the pattern with which distances grow differs between the four quadrants outlined in figure 4.7.

If the destination lies in quadrants 1 or 4, a route may be constructed as if in a hexagonal mesh topology. If the destination lies in quadrants 2 or 3, however, the algorithm tests whether taking a mesh-like vector within the quadrant or

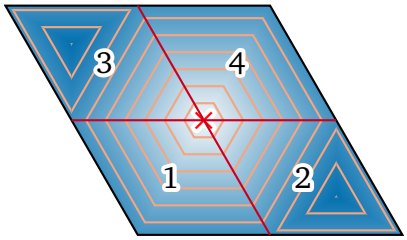


Figure 4.7: The four quadrants defined by the XYZ-protocol.

wrapping-around either the X or Y dimensions yields the shortest vector.

By comparison with the twelve vectors considered by the INSEE method, the XYZ-protocol considers just one for destinations in quadrant 1 or 4 and no more than three in quadrant 2 or 3. Though the XYZ-protocol can be computed more cheaply than the INSEE method, it does not produce valid shortest path vectors for hexagonal torus topologies with aspect ratios other than 1 : 1.

4.3 The Irregular Quadrant (IQ) method

In this section I propose a new technique for finding shortest path vectors in hexagonal torus topologies called the Irregular Quadrant (IQ) method and compare its performance with existing techniques.

4.3.1 Computing shortest path vectors

Consider the problem of finding a shortest path vector from (0, 0, 0), at the bottom-left, to a node somewhere else in a hexagonal torus topology.

Figure 4.8 illustrates how hexagonal torus topologies of various aspect ratios may be partitioned into four *irregular quadrants*. These quadrants are defined according to which axes are wrapped around by the shortest path vectors reaching them. The irregular quadrants correspond to locations reachable by shortest path vectors which:

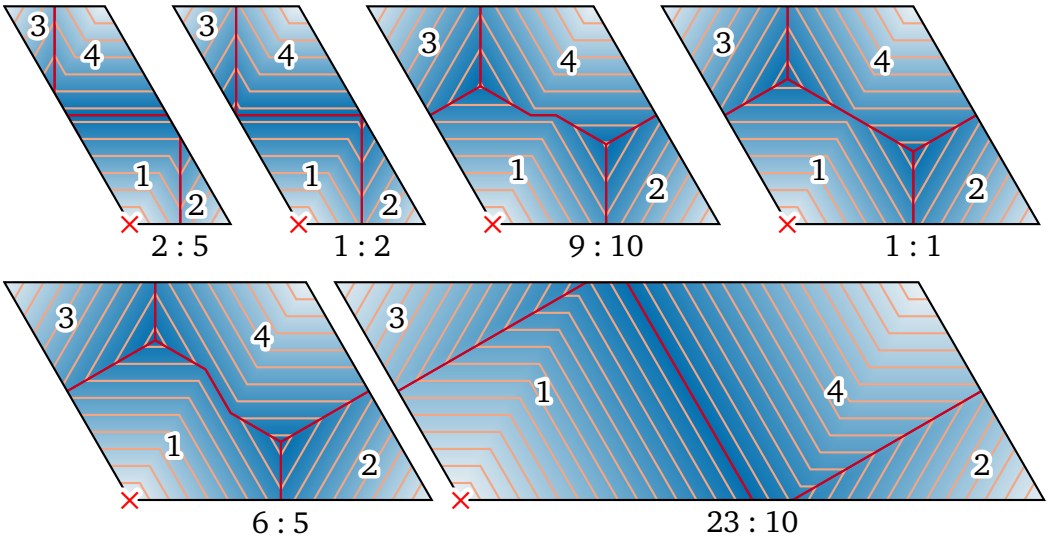


Figure 4.8: Hexagonal torus topologies of various aspect ratios divided into irregular quadrants in which a particular pair of dimensions are wrapped-around.

1. Do not wrap
2. Wrap around X only
3. Wrap around Y only
4. Wrap around both X and Y

Within each irregular quadrant, we observe that shortest path vectors are constrained to using only certain directions:

1. Only X^+, Y^+ and Z^-.
2. Only X^- and Y^+
3. Only X^+ and Y^-
4. Only X^-, Y^- and Z^+.

Given the topology is of width W and height H and the destination node's 2D mesh coordinates are (Δ_X, Δ_Y) we can define the shortest path vector within each irregular quadrant as:

1. minimiseVector$(\Delta_X, \Delta_Y, 0)$
2. minimiseVector$(-(W - \Delta_X), \Delta_Y, 0)$
3. minimiseVector$(\Delta_X, -(H - \Delta_Y), 0)$
4. minimiseVector$(-(W - \Delta_X), -(H - \Delta_Y), 0)$

Since we know that $0 \leq \Delta_X < W$ and $0 \leq \Delta_Y < H$, we can simplify the expressions for the magnitude of each of the above vectors. This yields four expressions giving the magnitude of a minimal vector as if in each irregular quadrant:

1. $\max(\Delta_X, \Delta_Y)$
2. $(W - \Delta_X) + \Delta_Y$
3. $\Delta_X + (H - \Delta_Y)$
4. $\max(W - \Delta_X, H - \Delta_Y)$

From these expressions we can determine which irregular quadrant the destination is in by finding the expression with the minimum value. With this information, the shortest path can be found by minimising the vector associated with that irregular quadrant. Figure 4.9 shows a simple Python implementation of the IQ method.

Compared with the INSEE method, this technique requires fewer cases to be considered (four rather than twelve) making it cheaper to compute.

```python
def iq_method(src, dst, w, h):
    """Get a shortest path vector from src to dst in a
    hexagonal torus with dimensions w and h. For example::

        >>> iq_method((1,2,0), (5,6,1), 10, 10)
        (0, 0, -3)
    """
    # Break-apart position vectors
    sx, sy, sz = src
    dx, dy, dz = dst

    # Convert to 2D mesh coordinates
    sx, sy, sz = sx-sz, sy-sz, 0
    dx, dy, dz = dx-dz, dy-dz, 0

    # Get the 2D vector from source to destination using
    # only positive coordinates.
    dx = (dx - sx) % w
    dy = (dy - sy) % h

    # Consider each irregular quadrant
    #             Distance           Non-minimised Vector
    quadrants = [(max(dx, dy),      (dx, dy)),
                 ((w-dx)+dy,        (-(w-dx), dy)),
                 (dx+(h-dy),        (dx, -(h-dy))),
                 (max(w-dx, h-dy), (-(w-dx), -(h-dy)))]

    # Select the quadrant with the shortest path (ties
    # broken arbitrarily)
    distance, (dx, dy) = min(quadrants)

    # Return a minimised hexagonal vector
    median = sorted((dx, dy, 0))[1]
    return (dx-median, dy-median, 0-median)
```

Figure 4.9: A Python implementation of the IQ Method.

Unlike the four quadrants defined by the XYZ-protocol, the irregular quadrants defined by the IQ method correspond exactly to a particular shortest path vector. This means that, once the irregular quadrant a point lies in has been discovered, the shortest path vector can be calculated directly without considering multiple options. Though the boundaries between the four irregular quadrants are more complex than the quadrants of the XYZ-protocol, it is only marginally more expensive to discover which irregular quadrant a point lies in. In addition, the irregular quadrants retain their meaning across different aspect ratios making the IQ method suitable for any hexagonal torus topology.

4.3.2 Computing distances

The four vector magnitude expressions defined by the IQ method can also be combined to produce a compact expression of the distance between two points in a hexagonal torus topology:

$$\begin{aligned}\|\,\text{shortestPathVector}(\Delta_X, \Delta_Y, W, H)\| = \min(\,&\max(\Delta_X, \Delta_Y),\\ &(W - \Delta_X) + \Delta_Y,\\ &\Delta_X + (H - \Delta_Y),\\ &\max(W - \Delta_X, H - \Delta_Y))\end{aligned}$$

This expression has also been derived independently from a graph-theoretic study of Cayley graphs, of which the hexagonal torus topology is a special case, by Xiao and Parhami [XP04].

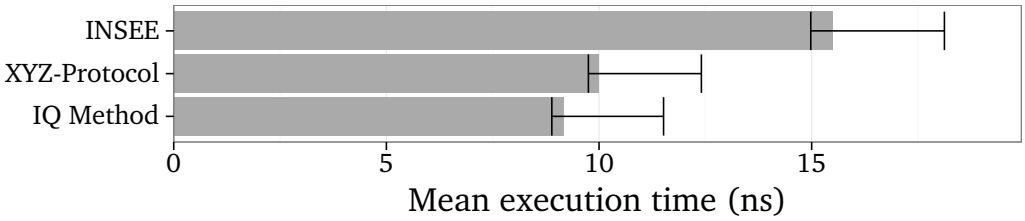


Figure 4.10: Mean execution time of each shortest path vector function. Error bars indicate the range of the mean execution times of the fifty 3-billion-combination experiments.

4.3.3 Computational efficiency

Since computing shortest path vectors forms an integral part of the kernel of many routing algorithms, the running time of the function chosen can be an important consideration.

To compare the performance of the three shortest path vector functions presented, the execution time of a C implementation of each technique was measured. The C implementation of the INSEE method is taken directly from the INSEE source code [NLMA+09]. The C implementation of the XYZ-Protocol is a straight translation of the published pseudocode [HD15].

Each function was called approximately 3 billion times: once for every pair of source and destination nodes in a 240 × 240 node hexagonal torus topology. The total execution time is then divided by the number of calls giving an average execution time. This experiment was repeated 50 times and the overall average execution time of each shortest path vector function was recorded. The repeats of the experiments were spread across a cluster of idle workstations with 3.10 GHz Intel Core-i5-2400 CPUs. The function implementations were compiled with GCC 5.3.0 with optimisations enabled (`-O2`).

Figure 4.10 shows the execution times of each shortest path vector function. From these results we can see that the INSEE method is slower than the other techniques. Although the XYZ-protocol and IQ method have similar per-

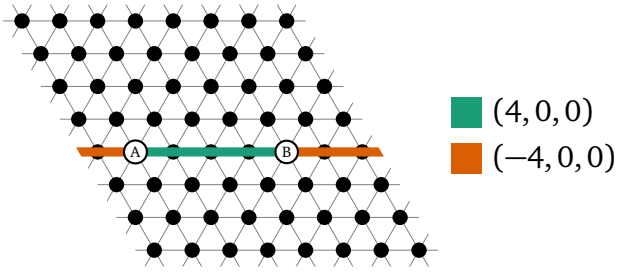


Figure 4.11: Two distinct shortest path vectors in a hexagonal torus.

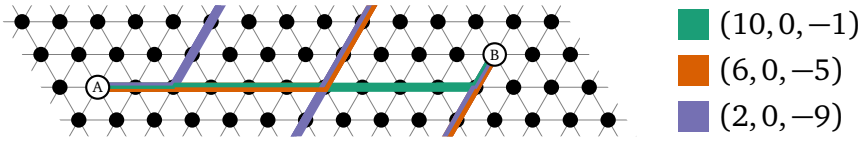


Figure 4.12: Distinct shortest path vectors between two points, all with magnitude 11.

formance, because the IQ method works for any hexagonal torus topology it is the better candidate for use in new applications.

4.4 Generating all shortest path vectors

In odd-sized non-hexagonal and 1 : 1 hexagonal torus topologies there is exactly one distinct shortest path vector between any two points (though many routes may be defined from it). In even-sized topologies there may be two distinct shortest path vectors between nodes exactly half the length of an axis away as in figure 4.11. The INSEE and IQ methods may generate either vector depending on how ties are broken in their implementation.

Unlike non-hexagonal topologies, when the aspect ratio of a hexagonal topology is not 1 : 1, some points may have many distinct (but equal magnitude) shortest path vectors. For example, figure 4.12 illustrates three distinct shortest path vectors between an example pair of nodes. None of the shortest path vector functions discussed will generate all possible vectors in this

situation, potentially limiting the choices available to routing algorithms. To address this shortcoming, I propose a formula which enumerates every shortest path vector between a pair of points in a hexagonal torus.

In a $W \times H$ hexagonal torus topology, starting from any node the vector $(0, H, 0)$ straightforwardly wraps once around the Y axis arriving back at the node it started from. The vector $(1, 1, 1)$ also returns to its starting point as described in appendix B. As a consequence $(0, H, 0) - H \cdot (1, 1, 1) = (-H, 0, -H)$ must also be a vector which leads back to where it started. Adding this vector to a shortest path vector of the form $(x, 0, z)$ results in a new shortest path vector if $x \geq H$ and $z \leq 0$.

For example, the vector $(10, 0, -1)$ (magnitude 11) in figure 4.12 can be added with $(-4, 0, -4)$ yielding $(6, 0, -5)$ (also magnitude 11) which is still a shortest path vector between the two labelled nodes. Since $(6, 0, -5)$ still meets the criteria defined above ($6 \geq 4$ and $-5 \leq 0$), we can add $(-4, 0, -4)$ again yielding another shortest path vector $(2, 0, -9)$. This new vector does not meet the requirement that $x \geq H$ (since $2 \ngeq 4$) and so no further shortest path vectors can be produced.

More formally, a shortest path vector $(x, 0, z)$ may be converted into another shortest path vector $(x', 0, z')$ using the following formula:

$$(x', 0, z') = (x, 0, z) - \left(\text{trunc}\left(\frac{z}{H}\right) + n\right)(H, 0, H)$$

where

$$\left\{ n \in \mathbb{Z} \;\middle|\; 0 \leq n \leq \left\lfloor \frac{|x| + |z|}{H} \right\rfloor \right\}$$

and trunc$(\cdots)$ is the truncation operator which rounds towards zero to the nearest integer.

A complementary formula may be derived based on the related observation

that the vector $(W, 0, 0)$ results in no movement:

$$(0, y', z') = (0, y, z) - \left(\text{trunc}\left(\frac{z}{W}\right) + n\right)(0, W, W)$$

where

$$\left\{ n \in \mathbb{Z} \;\middle|\; 0 \leq n \leq \left\lfloor \frac{|y| + |z|}{W} \right\rfloor \right\}$$

Using these two expressions for wide and tall topologies, respectively, all possible distinct shortest path vectors between any two points may be found.

4.5 Conclusions

The calculation of shortest path vectors in mesh and torus topologies is at the heart of many routing algorithms. In this chapter, I introduced a new technique for computing shortest path vectors in hexagonal toruses which generalises to any hexagonal torus topology while executing more quickly than existing approaches. In addition, I described a formula for enumerating the distinct shortest path vectors between points in hexagonal torus topologies. Unlike previous work, this allows routing algorithms greater freedom by providing a choice of shortest path vectors between some points. These contributions demonstrate that key operations such as computing distances and shortest path vectors in hexagonal toruses can be efficient and exhaustive just as in non-hexagonal torus topologies.

Chapter 5

Fault tolerant routing for SpiNNaker

In the previous chapter I developed an efficient technique for finding shortest path vectors in hexagonal torus topologies. These vectors are used as the basis for many routing algorithms, notably including all published SpiNNaker-specific algorithms. These routing algorithms, however, do not tolerate faults which are a typical hazard in any large-scale system. In this chapter, I introduce PGS repair, a post-processing step for existing routing algorithms which employs graph search to alter fault-afflicted routes in order to avoid faulty links and nodes.

In SpiNNaker, the vast majority of network faults appear to be the result of manufacturing defects which are static and known of before routing time. Though transient faults can occur and new faults may develop as the hardware ages. Because the large SpiNNaker prototype system constructed in chapter 3 is still undergoing initial commissioning, only limited data is available on its reliability in practice. During my own previous experimental work on the $^{1}/_{10}{}^{\text{th}}$ scale SpiNNaker prototype with 5760 chips (103 680 cores), test programs ex-

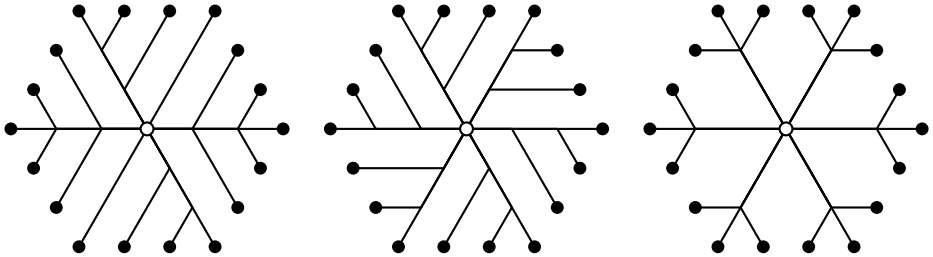
ecuting over the course of two to three days did not encounter any transient network faults. As a consequence, for many short and medium duration applications, static fault tolerance measures alone may be sufficient.

Numerous heuristic-based fault tolerant routing algorithms exist which target different network topologies and router architectures. Unfortunately, as I will show, these algorithms are not portable and rely on, or attempt to work around, specific features of their target network architecture. In particular, existing work is dominated by the challenge of developing routing schemes which avoid or resolve network deadlocks [DT04, chapter 14]. Due to SpiNNaker's unconventional use of timeout-based flow-control, it is not subject to the routing restrictions present in other architectures intended to cope with deadlocks. The post-processing algorithm proposed in this chapter relies on SpiNNaker's built-in deadlock avoidance capability to enable it to generate arbitrary routes to avoid faults.

To assess the performance of the proposed routing technique, a number of experiments have been performed using both software models and SpiNNaker hardware. For realistic fault rates, the proposed fault tolerant routing algorithm incurs negligible execution time overhead during route generation and only a small increase in network congestion and routing table usage.

5.1 Related work

In this section I discuss the techniques used by the fault intolerant multicast routing algorithms developed for SpiNNaker to date. I then explore the techniques employed by other architectures to achieve fault tolerance and consider their suitability for use with SpiNNaker.



(a) DOR (b) RTOR (c) LDFR

Figure 5.1: Example multicast routes produced by merging together unicast routes from a central source node to each destination node.

5.1.1 Multicast routing in SpiNNaker

Various fault intolerant multicast routing algorithms have been proposed and evaluated specifically in the context of the SpiNNaker architecture.

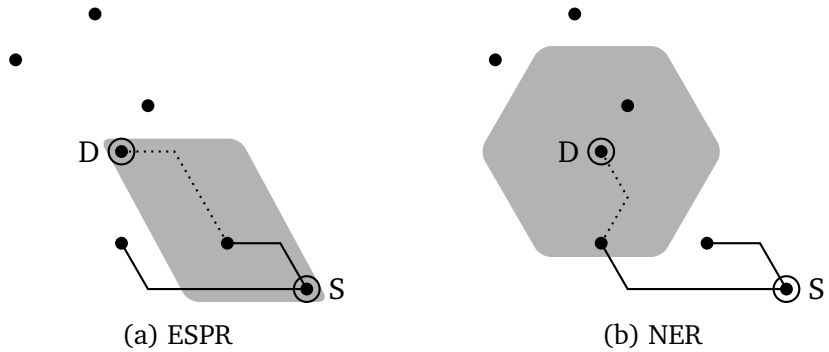
In 2012, Davies *et al.* evaluated the use of three common torus routing algorithms in SpiNNaker [DNGF12]. In each of the algorithms considered, multicast routes were constructed by merging together unicast routes from the source to each sink. The following unicast routing algorithms were considered:

Dimension Order Routing (DOR) Packets are routed along each dimension (e.g. X, Y and Z) in turn until no further hops are available in that dimension. The order in which the dimensions are traversed is fixed.

Right Turn Only Routing (RTOR) As in DOR except the dimension order is chosen so that routes only contain right-turns.

Longest Dimension First Routing (LDFR) As in DOR except the dimension order is chosen in descending order of number of hops in each dimension.

Figure 5.1 illustrates an example multicast route from a central source point to a number of sinks located along a hexagonal perimeter. The authors concluded that, of the three algorithms, multicast routes built using LDFR delivered the most balanced and efficient use of network resources.



(a) ESPR (b) NER

Figure 5.2: The ESPR and NER algorithms attempt to connect the node marked 'D' to the closest node in the shaded region which is connected to the source node, 'S'. If no connected node is found in the shaded region, the LDFR route is taken to 'S'. The dotted line indicates the route chosen from 'D'.

In 2014, Navaridas *et al.* introduced two new algorithms, 'Enhanced Shortest Path Routing' (ESPR) and 'Neighbourhood Exploring Routing' (NER) which are explicitly designed to generate efficient multicast routes [NLP+15].

In both ESPR and NER, the algorithm considers each sink vertex in turn, starting with those nearest to the source. For each vertex the algorithm attempts to find a nearby vertex to which a route already exists. If a connected vertex is found, an LDFR route from that vertex is added. If no nearby vertices are found, the algorithm inserts a LDFR route from the source vertex. By producing routes which connect to already-connected nearby vertices, ESPR and NER hope to reduce the number of long-distance routes from the source.

ESPR and NER differ in the way they search for nearby vertices. In ESPR, the algorithm searches within a parallelogram drawn to enclose the current and source vertices as illustrated in figure 5.2a. The NER algorithm searches for vertices within a fixed distance of the current vertex as illustrated in figure 5.2b.

The NER algorithm was found to produce higher quality multicast routes than both ESPR and the simpler algorithms described in earlier work. Unfortunately, as in previous work, this algorithm cannot tolerate faults.

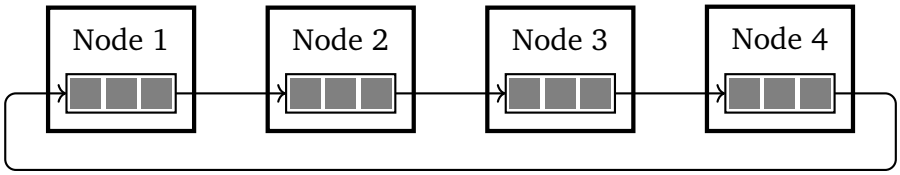


Figure 5.3: A deadlock in a ring network where each node is waiting for the next to accept a packet before accepting any further packets.

5.1.2 Fault tolerant routing

Numerous fault tolerant routing algorithms have been proposed for supercomputer networks but they are largely constrained by the need to maintain deadlock freedom. Since SpiNNaker's routers employ a timeout based deadlock-breaking strategy, much of this effort is unnecessary in SpiNNaker. As we see below, this frequently renders existing fault tolerant routing algorithms unnecessarily complex and inflexible making them unable to take advantage of the efficiency savings achieved by algorithms such as NER.

Deadlocks occur in a network when a cyclic dependency on any resource develops. For example, as illustrated in figure 5.3, in a ring network a deadlock may occur when every node is waiting for the next node to accept a packet before accepting new packets from the previous node.

To prevent deadlocks, combinations of router microarchitectural features and routing restrictions are often employed. For example, a simple deadlock-free (unicast) routing algorithm for mesh and torus networks mandates the use of DOR [DT04, chapter 14]. Packets travelling in a positive direction along each axis take priority over those travelling in a negative direction. Packets travelling along the X axis take priority over those travelling along the Y axis. Given these rules it is possible to define a total ordering on all hops in the network. Figure 5.4 illustrates a 3×3 mesh network whose hops have been numbered according to the total ordering defined above. Any 'X-then-Y' DOR

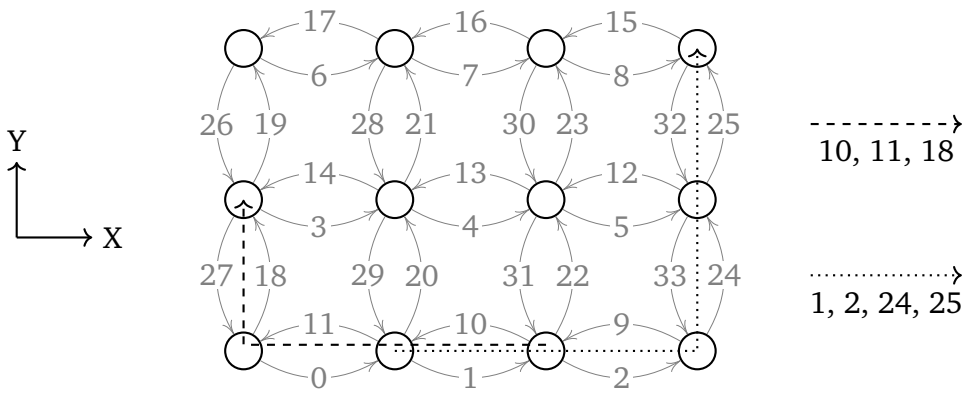


Figure 5.4: Deadlock-free routing of two example routes using DOR in a 2D mesh topology. Lower-numbered hops have higher priority. The numbers of the hops taken by each route are given on the right. (Figure based on [DT04, figure 14.12].)

route through this network results in the use of hops labelled with strictly increasing numbers. As a result, no cyclic dependencies (and thus no deadlocks) may occur. So long as incoming packets destined for a node are accepted from the network upon arrival, forward progress can always be made and thus deadlock is prevented.

Unfortunately, the routing restrictions imposed to ensure deadlock freedom can result in fault-intolerant routing. In the case of the DOR example, if the node at the bottom-right corner of the figure was faulty, the dotted example route would be blocked as no alternative routes are allowed. Although some routing schemes may feature more 'relaxed' routing rules [RMF+09], these may still ultimately not tolerate all faults.

The 3D torus based BlueGene/L supercomputer [AAA+02] notably makes use of DOR to ensure deadlock freedom while implementing fault tolerance by disabling so-called 'lamb' nodes. Lamb nodes are nodes which, though otherwise healthy, could potentially have their DOR routes blocked by a fault. This principle is illustrated in the 2D topology shown in figure 5.5 where a single

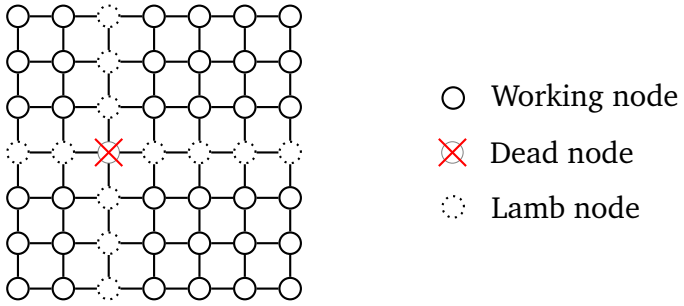


Figure 5.5: 'Lamb' nodes may be disabled to ensure DOR will never encounter a fault.

node has died. Nodes in the same row and column as the dead node are turned into 'lamb' nodes and are configured to forward packets but otherwise take no part in running applications. DOR routes between the remaining nodes will never pass through the dead node thus avoiding the fault while maintaining normal performance. This approach trades off wasted compute capacity for maintaining network performance in the presence of faults. When faults are rare and easily repaired this trade-off can be worthwhile. In SpiNNaker, such fine-grained replacement of nodes is impractical since each chip constitutes a node and therefore a whole board of 48 chips must be replaced to repair a single fault.

Other routing algorithms proposed for BlueGene attempt to avoid the need for lamb nodes by generating routes which reach their destination via a 'proxy' node [GDF+04]. In this scheme, a pair of nodes DOR route their messages via a 'proxy' node. This node is chosen so that routes to and from the proxy are not obstructed by the fault. This approach relieves the need to sacrifice 'lamb' nodes at the expense of adding to the workload of the proxy nodes and increasing routing complexity.

Alternative routing strategies may take a hybrid approach whereby an efficient but fault intolerant routing algorithm is used which falls back on a less

efficient, fault tolerant routing algorithm when faults are encountered. For example, the Immucube network architecture employs three virtual networks which operate independently over the same physical links [PG07]. Initially, messages are routed in the first virtual network using a high-performance routing scheme which may deadlock or be blocked by faults. If a deadlock occurs, the deadlocked packet is moved into the second virtual network in which packets are routed using a less efficient deadlock-free algorithm. Finally, if packets in the first and second virtual networks encounter a fault they are moved onto the third virtual network. The third virtual network forms a ring network which eventually reaches every node in the network, though potentially with a high performance penalty.

Instead of providing multiple routing rules which apply globally, some hybrid routing schemes divide the system up into regions in which different routing rules are used which avoid the particular patterns of faults in those areas [MFD+06, BC95]. As in other hybrid approaches, fault tolerance is usually achieved through the use of a relatively inefficient routing algorithm, though their use is constrained to a single region of the system.

Other routing algorithms are distributed and use only local information along with limited information from their peers to generate routes [FDC+09]. In SpiNNaker applications, route generation is typically carried out centrally, allowing algorithms such as ESPR and NER to take advantage of global information when producing multicast routes. Since routes in SpiNNaker applications are often static, using a centralised routing algorithm to produce high quality routes 'up front' may be favourable over potentially faster but lower quality distributed algorithms.

Algorithms for very different network architectures also tend to be a poor fit for SpiNNaker. IP networks, for example, must contend with a dynamic and

irregular network topology and so the trade-offs made in their routing algorithms are a poor match for the applications considered here. Similarly work focusing on architectures whose topologies differ significantly from a torus, for example data centre networks [GWT+08, LYYG12], are not general enough to be of use.

5.2 Partial Graph Search (PGS) repair

As we have seen, existing multicast routing algorithms for SpiNNaker are designed to make efficient use of the system's network but are not able to avoid faults. The hybrid routing approach used by Immucube allows an existing, efficient routing algorithm to be used while falling back on other routes when faults are encountered. In this section I describe a novel hybrid routing algorithm based on a post-processing scheme called Partial Graph Search (PGS) repair. By relying on SpiNNaker's timeout-based deadlock avoidance scheme, PGS repair is not hampered by the routing restrictions imposed by other architectures. For example, expensive router architectural features, such as the multiple virtual networks used by Immucube, are not required. In addition, the routes taken to avoid faults are not constrained to inefficient paths, such as system-wide ring networks, to ensure deadlock freedom.

The PGS repair technique uses a graph search algorithm to find alternative routes around faulty nodes and to repair fault-afflicted routes generated by a fault intolerant routing algorithm such as NER. Though graph search algorithms are generally shunned in network routing due to their high computational complexity, their execution time is often bounded by $O(H^D)$ where H is the number of hops in the generated route and D is the number of dimensions in the network. Since H is likely to be small because faults are rare and isolated

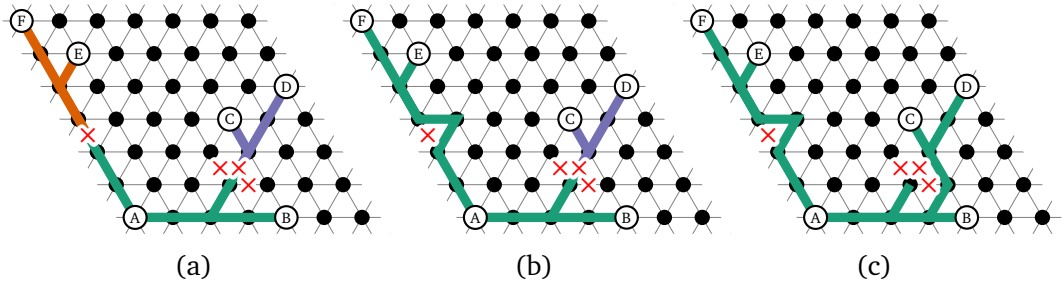


Figure 5.6: PGS repair example showing a disconnected multicast route from A to B, C, D, E and F. × indicates a broken link.

in real systems [GBC+05, AFKR12], the execution time of the graph search will also be small in practice.

The PGS repair technique proceeds as follows. A route is initially generated by a fault intolerant routing algorithm such as NER. If no faults are encountered by this route, the algorithm terminates immediately. Otherwise the algorithm proceeds to repair the route as follows:

The routing tree produced by the underlying routing algorithm is broken into subtrees wherever it attempts to route through a broken link or node. Each subtree is then assigned a unique 'colour', as illustrated in figure 5.6a. From each disconnected subtree's root in turn, a graph search is performed to find a short, fault-free route to a subtree of any other colour. Once another subtree has been found, the route discovered by the graph search is added and the subtree is coloured to match the tree from which the graph search started from.

For example in figure 5.6b, a path from the root of the subtree containing nodes E and F is found which connects it to the subtree rooted at A. In figure 5.6c a path is also found connecting the subtree containing nodes C and D back to the subtree rooted at node A. At this point only one tree remains – the fully connected route.

If the routing tree was broken into $N+1$ subtrees by faults there will be N

subtrees disconnected from the root node. Each of the N graph searches performed connects two disconnected subtrees reducing the number of subtrees by 1 each time. After N iterations, therefore, exactly one tree remains which connects every node in the original routing tree resulting in a multicast route not obstructed by faults.

Because SpiNNaker's network architecture ensures deadlock freedom regardless of the routing pattern used, the graph search algorithm is free to use any route to avoid dead links, reducing the overhead introduced by avoiding faults.

5.3 Evaluation and results

The PGS repair technique is, by design, able to work around all possible fault patterns (excluding those which completely disconnect entire regions of the network). As a consequence, my evaluation focuses on the impact on performance that PGS repair imposes rather than its ability to avoid faults. Specifically, I concentrate on the impact of PGS repair on three figures of merit: algorithm execution time, routing table utilisation, and network throughput.

5.3.1 Traffic patterns

Two standard benchmark multicast traffic patterns are considered which have been used in previous research into SpiNNaker's network:

Uniform Destinations are chosen with uniform probability anywhere in the machine. This traffic pattern is widely used in networks research [DT04, chapter 23] and is representative of applications with either dense communication requirements or poor placement quality (see chapter 6).

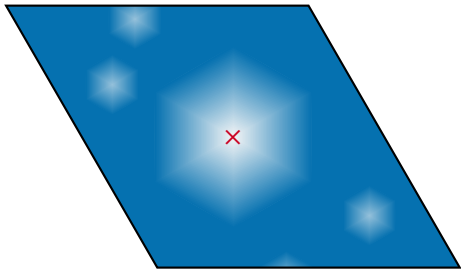


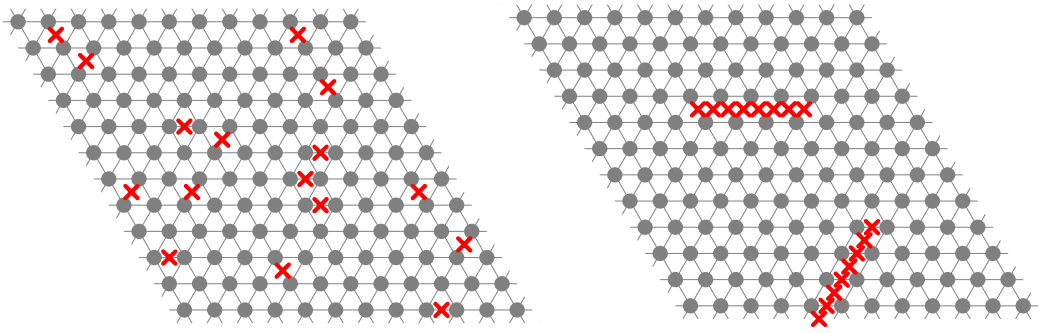
Figure 5.7: An example of a 3-centroid traffic distribution. Traffic is sent by the node at the position marked with an × to nodes chosen randomly with nodes in lighter areas being selected with a higher probability.

3-centroids Destinations are clustered around the sending node with an 85% probability or one of 3 randomly chosen 'centroids' with a 5% probability each as illustrated in figure 5.7. Destinations are distributed around the sender or centroid according to a geometric distribution with more distant locations being less likely to be selected. This pattern was developed specifically to reproduce the traffic patterns found in the neural applications SpiNNaker is designed for [NLP+15].

There is little consensus on the 'fan out' of routes in real SpiNNaker applications. As we will see in the next chapter (§6.4.2), the fan outs of existing applications vary between unicast routes and multicast routes with almost one hundred sinks. Fan out does not have a significant impact on the performance of PGS repair and so I have simplified this evaluation by only considering routes with a single source node and sixteen sink nodes. This choice ensures that generated routes feature multicast branching ensuring PGS repair is suitable for multicast routing.

5.3.2 Fault models

Two different fault models are considered which are chosen to be representative of the faults found in real SpiNNaker systems:



(a) Uniform (b) HSS Link

Figure 5.8: The two link fault models considered.

Uniform Links are selected and disabled at random (figure 5.8a). This models failures resulting from isolated manufacturing defects or ageing in individual links.

HSS Link Groups of links corresponding with randomly selected High-Speed Serial (HSS) links between SpiNNaker boards are disabled (figure 5.8a). Each HSS link carries several chip-to-chip traffic flows via a single cable; problems result in a contiguous line of failures.

Though SpiNNaker-specific, the latter fault model is analogous to failure modes arising in other architectures where a single fault may render several links impassable in a single area.

Informal measurements of current large-scale SpiNNaker installations suggest that the link failure rate is approximately 0.03 % with failures due to both individual chip-to-chip links and board-to-board HSS links. Exact link failure statistics for commercial supercomputer installations are not widely available, however, published Mean-Time-Between-Failure (MTBF) figures coincidentally place an upper bound on link failure rates at a similar 0.03 % in one-year-old BlueGene/Q systems [CCWS11].

In addition to link faults, there is currently an undiagnosed problem with

the SDRAM packaged with around 1 % of SpiNNaker chips which renders them unusable. These faulty chips leave gaps in the network which dominate 'true' link faults making the effective link fault rate rise to 1 %.

Surprisingly, research into fault tolerant routing in supercomputers appears to focus on benchmarks with even higher fault rates which range from 3 % to as high as 7 % [HS04, GDF+04, MFD+06].

In this evaluation, fault rates ranging from 0.01 % to 5 % are considered to cover both realistic fault levels along with the more extreme cases considered in the literature.

5.3.3 Base routing algorithm

Since the PGS repair process may be used with any routing algorithm, all experiments have been conducted against routes initially generated by the NER routing algorithm since this algorithm is known to produce high quality routes for SpiNNaker's network [NLP+15].

5.3.4 Algorithm execution time

To assess the execution time overhead introduced by the PGS repair process, routing problems were generated for a 256×256 hexagonal torus topology with randomly generated faults. For each combination of fault pattern and traffic pattern, 10 000 route and fault sets were generated and the execution time of the NER and PGS repair algorithms recorded.

The NER and PGS repair algorithms were written in C and compiled with GCC 4.8.3 with `-O2` level optimisations and instances of the experiments were spread across a cluster of idle workstations with 3.10 GHz Intel Core-i5-2400 CPUs.

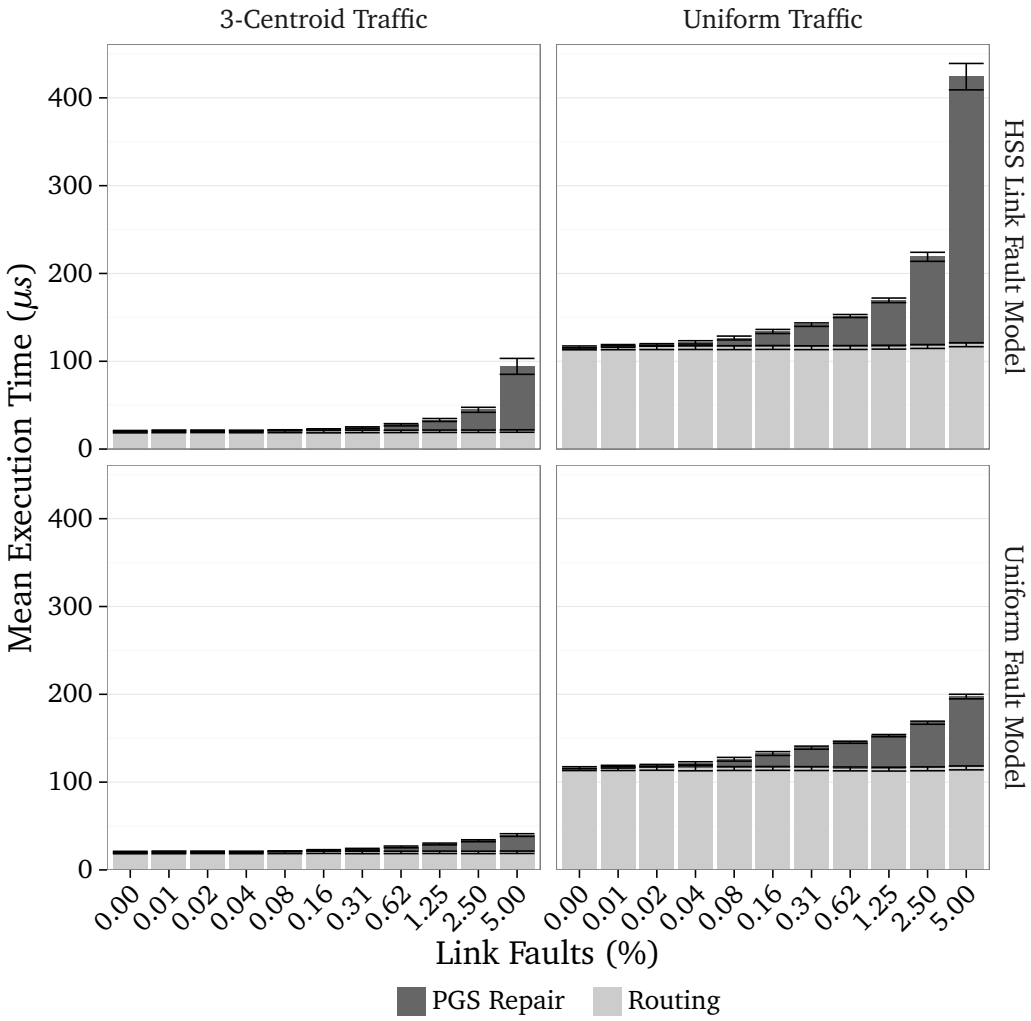


Figure 5.9: Mean execution time of NER routing and PGS repair. Bars are stacked and do not overlap. Error bars indicate 95% confidence interval.

Figure 5.9 shows the mean execution times for both the NER and PGS repair algorithms. In fault-free networks, the PGS repair post-processing step is not required and incurs no penalty. As the fault rate increases, the execution time of the algorithm grows proportionally for both fault and traffic models.

The HSS fault model results in longer execution times for the PGS repair process compared with an equivalent fault-density of uniform faults. Because the HSS fault model produces contiguous lines of faults, the PGS repair algorithm must construct a longer path to avoid the fault. Since the space explored by a graph algorithm grows with $O(H^2)$ in a 2D hexagonal torus topology, this increase in search distance has a significant impact on the execution time of the PGS repair process.

The centroid traffic pattern tends to result in routes with fewer hops than uniform traffic since route segments are often shared between destination nodes in the same centroid. Since the NER algorithm's execution time is strongly related to the number of hops in the output route, the execution time of the algorithm is greater for uniform traffic. Likewise the probability of a route encountering a fault, and thus PGS repair being invoked, increases with the number of hops in the route, hence PGS repair execution times are also reduced under centroid traffic.

5.3.5 Routing table usage

SpiNNaker uses a table based router with a fixed quantity of table entries in its multicast router. Algorithms such as NER attempt to exploit 'default routing' which allows routing table entries to be omitted on chips where a packet is forwarded in a straight line by generating routes which avoid turning corners. PGS repair, however, makes no such effort and thus may be liable to increase the number of routing table entries required.

To gain a realistic measure of routing table usage it is necessary to produce a range of routes for a single example network and generate routing tables accordingly. To enable a sufficiently large number of samples to be collected smaller networks with 48×48 nodes are considered in this experiment.

One thousand 48×48 node networks are produced according to the HSS link and uniform fault models. For each of these networks, two sets of $48 \times 48 \times 16 = 36\,864$ routes are generated: one using the centroid model and the other the uniform traffic model. This number of routes corresponds to one multicast route per application core in a 48×48 SpiNNaker system.

Routing table entries are inserted at the source node of each route, at each destination node and at every corner or fork, exploiting default routing to route packets along other parts of routes. This table generation scheme is widely used in SpiNNaker applications due to its simplicity and efficient use of routing table entries. Though more effective techniques exist for compressing routing tables for use in SpiNNaker's network [MHG16], their performance is strongly influenced by other factors such as routing key selection and so are not considered here.

For each network model and traffic pattern, the number of routing table entries required on each node is counted. The maximum count is then recorded since even if only one routing table is too large to fit in SpiNNaker's router the application cannot run.

Figure 5.10 shows the distributions of the largest routing table sizes for each fault and traffic model. For realistic fault rates (under 0.1 %), routing table sizes are not significantly impacted.

At a 1 % fault rate, routing table sizes grow little in the presence of uniform faults but in the presence of HSS link faults exceed routing table size limits. Since SpiNNaker's 1 % fault rate can be attributed to isolated, uniformly dis-

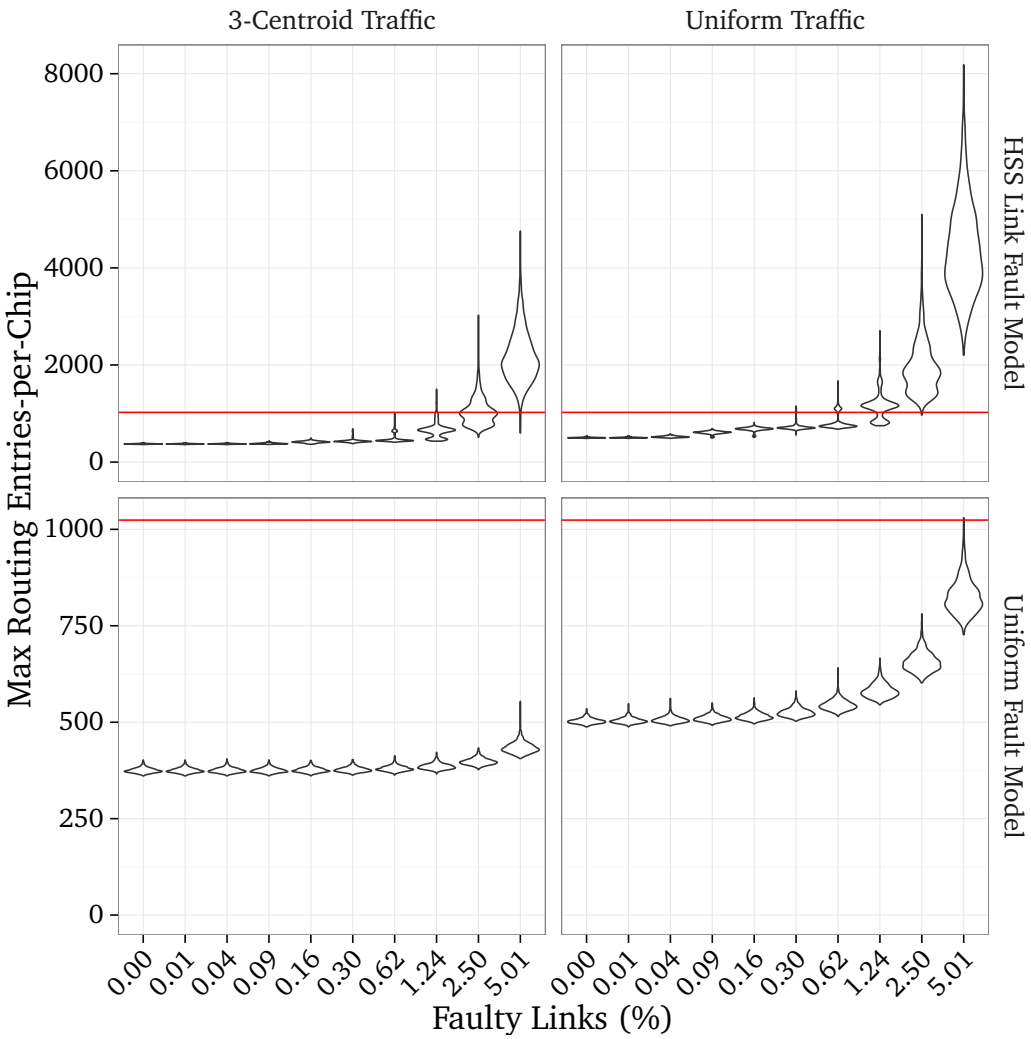
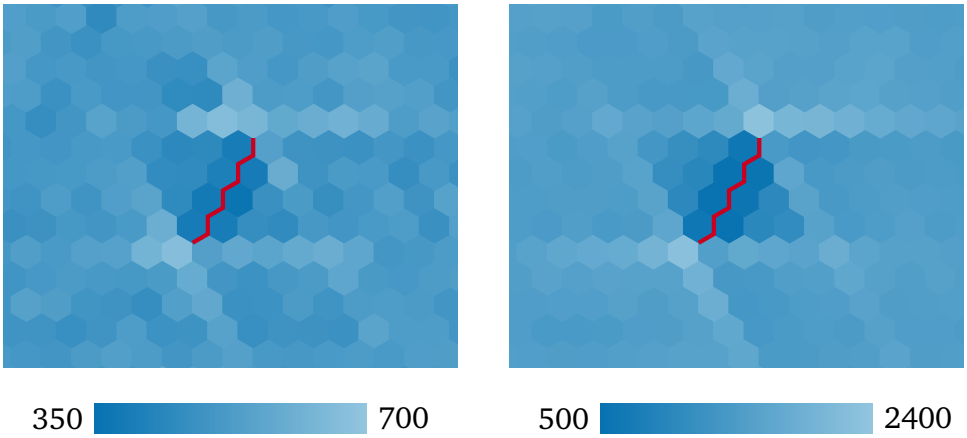


Figure 5.10: Violin plot showing the distribution of maximum routing table sizes for 1000 random networks. The red line at 1024 entries indicates the size of SpiNNaker's routing tables.



(a) Routing table entries (b) Routes passing through chip

Figure 5.11: The impact of a HSS link fault on routing table usage and congestion. Each hexagon represents a single chip, the red line indicates the chip-to-chip connections broken by the HSS link fault.

tributed SDRAM faults, the HSS link fault model is not representative of real faults at this fault rate. As such, I conclude that for the fault rates and fault patterns experienced by real SpiNNaker machines PGS repair has no significant impact on routing table usage.

As the fault rates climb to more extreme levels, routing table sizes grow more quickly under the HSS link failure model. This is because HSS link faults result in a large concentration of routes being re-routed around the same obstacle. Figure 5.11a shows how routing table sizes have doubled around a HSS link fault in one instance of the experiment. The peaks in routing table use are concentrated around the tips of the HSS link fault as a result of PGS repair always taking shortest path routes around obstacles.

5.3.6 Network congestion

To measure the impact of PGS repair on network congestion, two experiments were performed. In the first, the same model is used as in the routing table

usage experiments, using a simple metric to estimate network congestion, and thus throughput. The second experiment attempts to confirm the validity of the first using SpiNNaker hardware. A synthetic traffic generator running on SpiNNaker was used to determine network throughput in the presence of simulated faulty links. Because of the baseline fault rate of 1 % and limited routing table size in real SpiNNaker hardware, these experiments are more limited in scope and sample size.

Software model

For each of the network fault and traffic patterns described in the previous experiment, the number of times each link is used is counted. The number of routes passing through the most-used link is recorded, giving an indication of the peak congestion in the network. In general, network throughput is bounded by the slowest, or in this case, most congested link in the system [DT04, chapter 1] making this metric a suitable proxy for peak throughput.

The results are presented in figure 5.12 and follow the same trends as the results shown for routing table usage. Again, HSS link faults result in routes with the greatest congestion due to the concentration of routes finding shortest paths around an obstacle as seen in figure 5.11b.

SpiNNaker experiments

To verify the results produced by the software model, additional experiments have been carried out using SpiNNaker hardware. In these experiments a synthetic network traffic generator is loaded onto an 8×8 chip SpiNNaker machine partitioned from a larger SpiNNaker installation. Because these machines are subsections of larger torus topologies, they lack wrap-around links and form hexagonal mesh topologies. It was not possible to use larger topologies be-

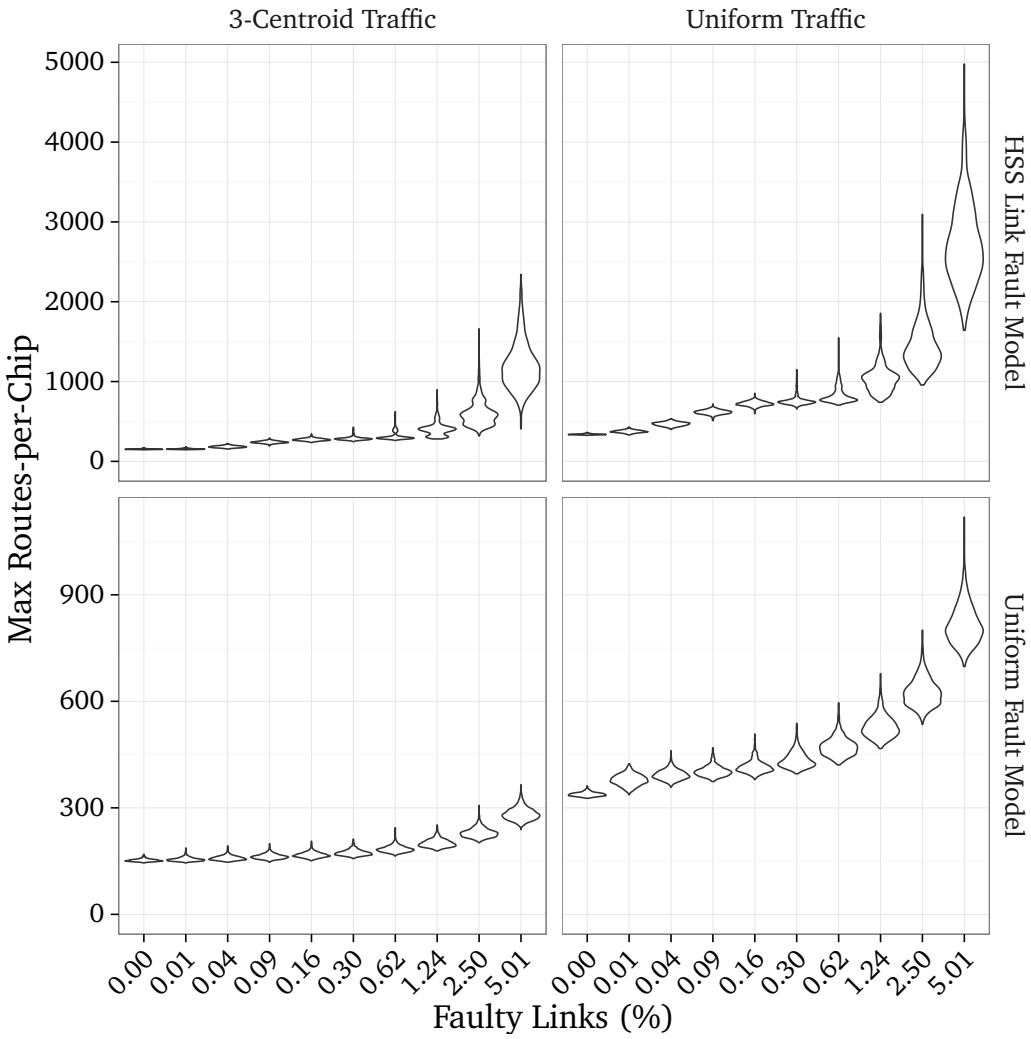


Figure 5.12: Violin plots showing the distribution of maximum routes-per-chip for 1000 random networks.

cause, at some of the fault rates and distributions considered, the 1024 routing table entries available in SpiNNaker are insufficient.

The traffic generation software running on SpiNNaker is configured to produce uniform and 3-centroid traffic patterns with routes generated by the NER and PGS repair algorithms. Simulated network faults are inserted according to the HSS link and uniform fault models in addition to those already present. Due to the baseline fault rate of 1 % these experiments only consider fault rates above this.

Packets are generated according to a Bernoulli distribution. The rate of packet injection is gradually increased across all routes until SpiNNaker's network becomes saturated and packets start being dropped. Once fewer than 99 % of packets arrive at their intended destinations, the network is deemed to have saturated and the peak throughput achieved prior to saturation is recorded. This experiment is repeated 100 times, each time with a different randomly generated network and on a randomly partitioned block of a larger SpiNNaker system.

Figure 5.13 confirms that, as suggested by the software model, increased fault rates result in reduced network throughput. HSS link faults also continue to result in greater performance segregation than uniform faults. Increasing the fault rate from 1% to 3% halves throughput for HSS link faults while in the presence of uniform faults throughput is only reduced by 10%.

In the experiments performed on SpiNNaker, the uniform and centroid traffic patterns differ little. This similarity is attributed to the small size of the network under test resulting in there being little difference between the shape of the traffic patterns at that scale.

Overall, the results gathered on SpiNNaker confirm the trends suggested by the software model.

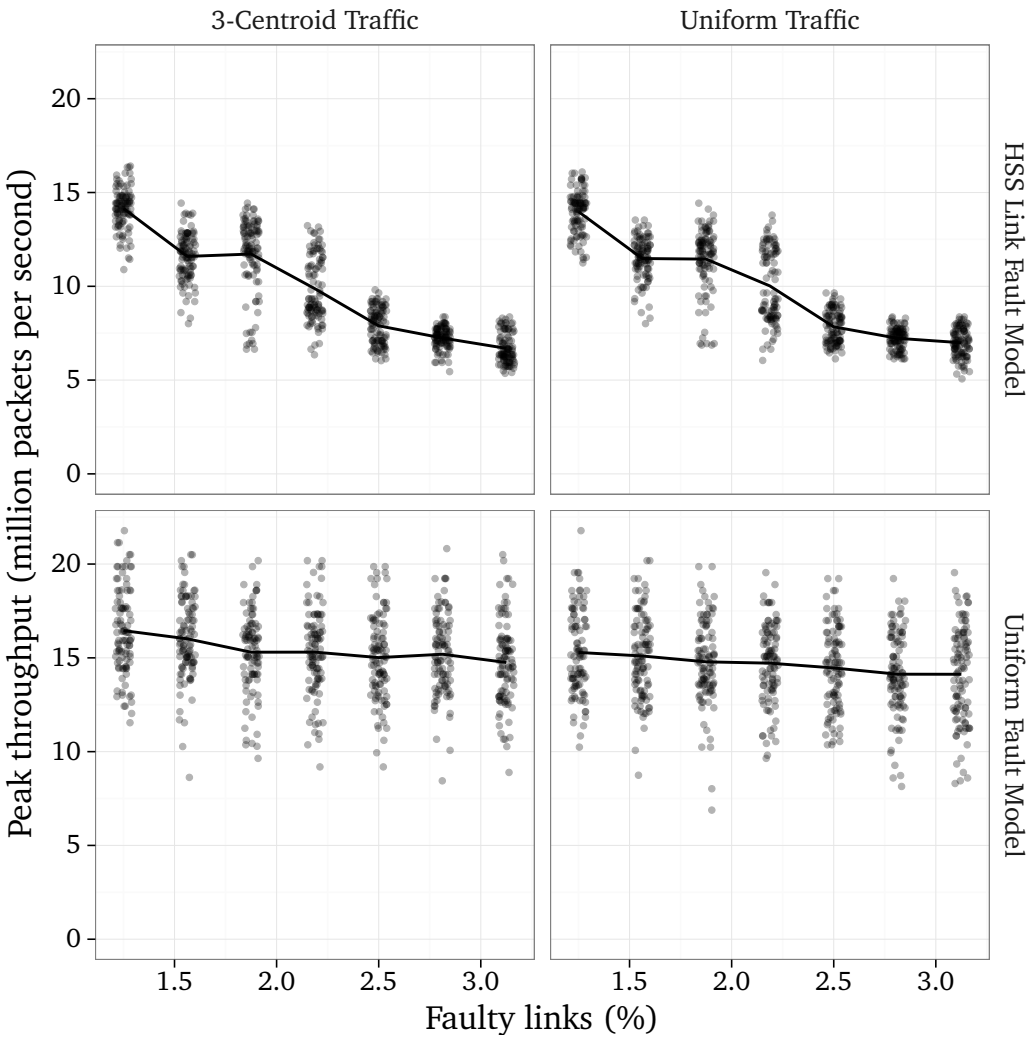


Figure 5.13: Maximum throughput achieved by 100 random traffic patterns running on SpiNNaker. The dots show the recorded result (jittered in the 'X' direction for clarity) and the lines show the mean.

5.4 Conclusions

Fault tolerance is a necessity in large-scale systems such as SpiNNaker and other supercomputers. Thanks to SpiNNaker's unconventional use of timeout-based deadlock avoidance, the expensive trade-offs made by routing algorithms for other architectures need not be made. The PGS repair algorithm introduced in this chapter is able to exploit the high quality routes generated by existing fault intolerant routing algorithms while handling faults with minimal routing and network overhead.

Unlike some other fault tolerant routing algorithms, PGS repair is able to work around arbitrary fault patterns when a route exists. In the presence of fault rates of up to 0.1 %, representative of real-world systems, the execution time, routing table use and network performance overhead introduced were found to be negligible. In the presence of more extreme fault rates of up to 1 %, representative of current SpiNNaker machines suffering from an unknown SDRAM issue, the overhead remains manageable. Overhead increased by 30 %, 11 % and 44 % for algorithm execution time, routing table usage and network performance respectively. As a result of these performance characteristics, PGS repair is a viable option for SpiNNaker applications.

At more extreme fault rates, not anticipated in real-world systems, PGS repair produces routes which are valid but incur much greater congestion and routing table overhead, particularly when faults are concentrated in one area. Future extensions to this algorithm might aim to reduce this overhead by producing longer and more varied routes around faults to even out load.

The success of the PGS repair algorithm demonstrates that SpiNNaker's unconventional deadlock avoidance scheme may warrant greater attention in future, larger scale systems in which more faults may be present. It may also be applicable to low-cost systems with less stringent quality controls.

Chapter 6

Placing applications in large SpiNNaker machines

In the previous chapter I tackled the problem of fault tolerant route generation for SpiNNaker's network. The centroid traffic pattern was used as an approximation of the expected network traffic generated by 'well behaved' neural network simulation software running on SpiNNaker. This traffic pattern exhibits locality, with most communication occurring between adjacent nodes. In reality, neural simulation applications are not conveniently specified geometrically but rather as abstract graphs of communicating neurons [DBE+08, EA04]. Although neural models often exhibit localised connectivity, this can only be exploited if they are laid out sensibly within a network. Producing optimal placements is an NP-complete problem [HS11].

In this chapter I evaluate the effectiveness of simulated annealing [KGV+83] for placement of SpiNNaker applications. Though simulated annealing fell out of favour for use in application placement in the 1990s, I argue that it is time to re-evaluate its suitability for modern placement problems. Looking to the field of chip design, where circuit placement problems have also been grow-

ing exponentially in scale for the past few decades, simulated annealing was considered ideal for placement problems whose sizes are comparable to that of modern supercomputer placement problems.

In experiments on existing SpiNNaker applications I find that simulated annealing achieves placement solutions substantially better than contemporary application placement techniques. In some cases simulated annealing is shown to make the difference between an application functioning correctly and not at all. Using larger scale synthetic benchmarks I have also demonstrated that my proof of concept simulated annealing based placement algorithm scales up to placement problems containing over one million vertices while achieving placement quality similar to an optimal placement solution.

I also build upon the work on hexagonal torus topologies earlier in this thesis to develop an efficient placement cost estimation function for use in the kernel of the annealing algorithm.

6.1 Placing neural simulations

At its most abstract, the application placement problem is that of finding a mapping from an application graph to a machine communication graph. Each vertex in the machine communication graph represents a node in the machine's network (e.g. a SpiNNaker chip) and each edge represents a physical connection between two vertices. Each vertex in the application graph represents some process which must be placed onto a vertex in the machine communication graph. (Multi-)edges between vertices in the application graph represent communication between those vertices (e.g. a flow of multicast network packets from one vertex to a number of others).

To give a more concrete explanation, consider the following example of

how a typical neural model defined using the PyNN or Nengo modelling tools might be expressed as an application graph. PyNN and Nengo define models in terms of 'populations' of neurons. A population may contain many thousands of neurons whose parameters are drawn from a user-defined distribution. The connectivity between neurons are coarsely defined in terms of 'projections' between populations and not on a neuron-by-neuron basis. The precise neuron-level connectivity is once again defined by a user-defined distribution.

To simulate a neural model as defined above, populations of neurons are automatically partitioned into 'subpopulations' containing a few hundred neurons such that each subpopulation's computational needs can be met by an individual SpiNNaker core. Spikes are transmitted via multicast packets which are multicast to every subpopulation targeted by a projection. Spikes are filtered on arrival for their relevance to neurons in a given subpopulation. This process simplifies network routing since only one set of routing entries is required per projection.

This type of application is represented by an application graph in which each vertex represents a core responsible for simulating the neurons in a particular subpopulation. Projections are represented by edges between the subpopulations they connect in the application graph.

A placement algorithm must assign subpopulations (vertices in the application graph) to chips (vertices in the machine communication graph) such that no chip's cores or SDRAM are over-allocated. A placement algorithm may also try to ensure that projections (edges in the application graph) are kept as short as possible within the machine to reduce the load on the network.

In principle it is possible to place neural simulations at a finer granularity. For example, populations could be broken into individual neurons and edges might connect on a neuron-by-neuron basis. Though the finer granular-

ity might allow more efficient packing of neurons into SpiNNaker's network it dramatically increases the problem size. As is shown later in this chapter, increased problem sizes could lead to much longer placement algorithm runtimes.

6.2 Related work

Attempts to address the problem of placing applications in supercomputer networks have utilised a variety of approaches ranging from simple greedy algorithms to sophisticated combinatorial optimisation techniques [JMT14]. In the parallel field of digital circuit placement for chip and FPGA design, placement algorithms have also received intense study. In this section I survey the work carried out on application placement before exploring chip design techniques to see what can be learnt from this perhaps more advanced field.

In the application placement literature, the placement problem is often referred to by the umbrella term 'mapping'. Unfortunately, 'mapping' is often used more broadly to include other tasks such as routing and application partitioning. To avoid ambiguity I use the term 'placement', as preferred by the chip and FPGA design communities, to refer specifically to the problem of assigning vertices in an application's communication graph to nodes in a machine's connectivity graph: the focus of this chapter.

6.2.1 Application placement algorithms

Below I consider several popular techniques employed by existing supercomputer applications ranging from application-specific manual placement to sophisticated general purpose algorithms.

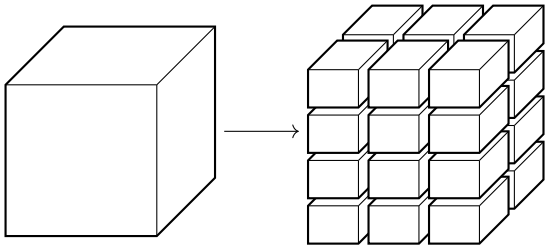


Figure 6.1: Example partitioning of a 3D volume to fit into a supercomputer with a $3 \times 4 \times 2$ torus or mesh topology.

Application-specific approaches (manual placement)

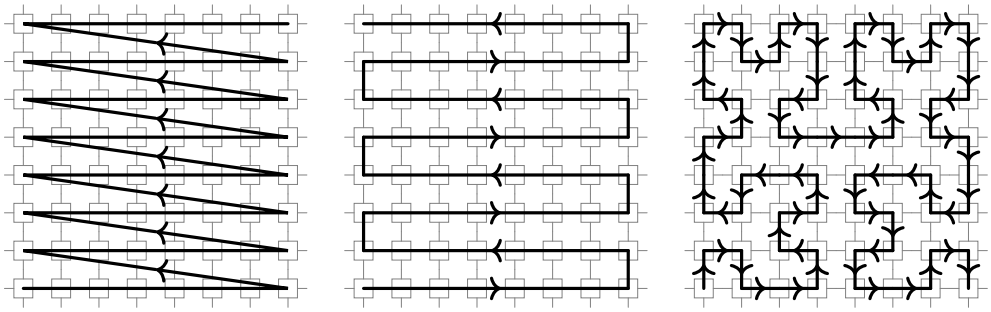
In certain applications such as finite element modelling [BMBL+13], a problem's structure defines a 'natural' placement of the problem onto the target machine. For example when simulating a 3D volume in a supercomputer with a $3 \times 4 \times 2$ 3D torus or mesh topology network, the modelled volume might be divided as in figure 6.1. Each cuboid in the model is then assigned to the corresponding node in the network topology.

Unfortunately, this technique only applies to a very narrow class of applications and is not applicable to many of the types of neural models SpiNNaker is designed to run.

Sequential placement

In the case where a placement solution is non-obvious, one of the simplest and most popular strategies is to apply a simple sequential placement algorithm. Sequential placement greedily places the vertices of an application's communication graph onto nodes. By carefully selecting the order in which vertices are picked and nodes assigned in the target architecture, this algorithm can produce reasonable quality placements.

Supercomputer management software such as SLURM [YJG03] and Blue Gene's system software [Gil14] by default naïvely iterate over vertices in an



(a) Row-order (b) Alternating (c) Hilbert curve

Figure 6.2: Space-filling curves in 2D mesh and torus topologies.

application communication graph in the order in which they are defined by the application. The nodes in the target machine are iterated over using a simple space-filling curve. Figure 6.2 illustrates three popular space filling curves.

The row-order (figure 6.2a) and alternating (figure 6.2b) curves illustrate 2D versions of the default node assignment orders provided by the SLURM and BlueGene systems. The Cray extensions to SLURM software provide a Hilbert curve [Hil91] node assignment order (figure 6.2c). The Hilbert curve specifically attempts to maintain the property that points that are close together along the length of the curve are also close together in the 2D mapping [MJFS01, Zum99]. Since the vertex iteration order is also chosen such that connected vertices tend to be closer together, this property may lead to improved placement solutions.

Figure 6.3 illustrates how the distance between locations along each space filling curve map to distances in the resulting 2D space. Both the row-order and alternating orderings result in distances oscillating between being closer and spread out in 2D space, just as the space filling curves oscillate from side-to-side. For the Hilbert curve, however, distances grow more monotonically[1]

[1]The growth in distance for Hilbert curves may appear slightly irregular but in fact forms a fractal as a consequence of the fractal construction of the Hilbert curve itself.

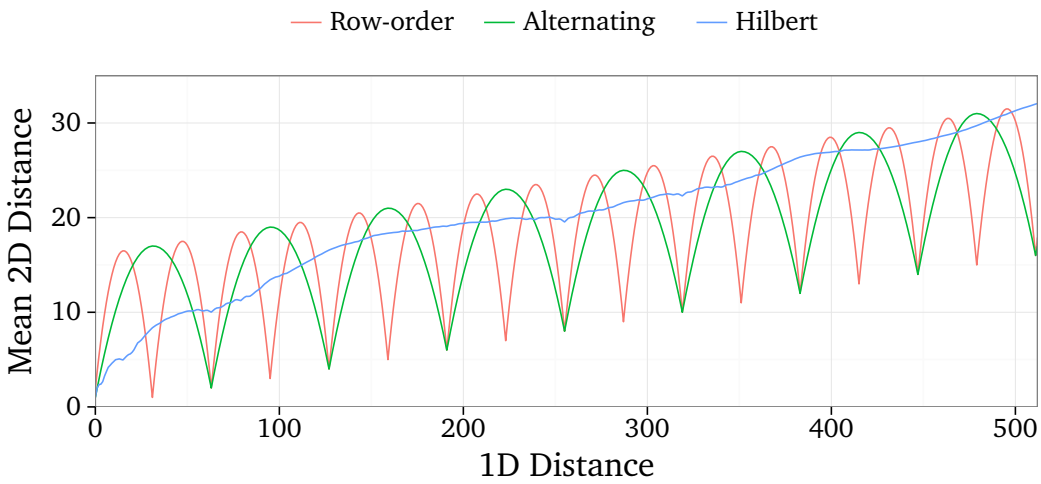
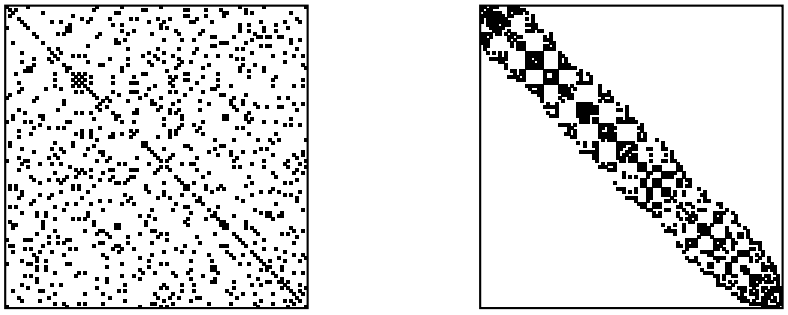


Figure 6.3: Relationship between distance along a space filling curve and between the 2D positions assigned. Results shown for Manhattan distances in a 32×32 2D mesh.

and at smaller 1D distances, 2D distances are often lower than in other space filling curves.

A number of algorithms have been proposed for automatically selecting good vertex and chip iteration orders, typically using a graph-traversal based heuristic. An example typical of this type of technique is 'graph similarity based mapping' described by Hoefler and Snir [HS11]. This algorithm exploits the Reverse Cuthill-McKee (RCM) algorithm [CM69] to select a linear ordering of vertices in which connected vertices tend to appear close together. Internally, the RCM algorithm uses a breadth-first-search-like algorithm to order vertices.

The RCM algorithm was originally designed as a tool for reducing the 'bandwidth' of matrices to improve the performance of linear algebra techniques. A matrix's bandwidth is the width of the strip of non-zero elements on its diagonal. Figure 6.4 illustrates how a sparse matrix is permuted by the RCM algorithm to produce a matrix with lower bandwidth. This technique may be applied to a graph by representing the graph as an adjacency matrix, M, where



(a) Original permutation (b) RCM permutation

Figure 6.4: Adjacency matrix representation of a graph before and after permutation by the RCM algorithm.

$M_{i,j}$ is w if node i is connected by an edge to node j with weight w and 0 otherwise. When a graph's adjacency matrix is permuted by the RCM algorithm, the connectivity of the graph remains unchanged but the vertex ordering changes. The ordering of vertices in a low-bandwidth adjacency matrix tends to keep connected vertices closer together in the ordering, making the ordering well suited for use with a sequential placement algorithm.

In small-scale and densely connected applications such as early neural simulations running on prototype SpiNNaker machines [GRDF10], these techniques have proven sufficient. As my experiments demonstrate later, however, sequential placement is insufficient for some SpiNNaker applications.

Simulated annealing

In the academic community, a number of attempts have been made to use sophisticated optimisation algorithms for the placement of applications. In 1985, Steele [Ste85] first proposed the use of simulated annealing for placing applications in the 6D torus topology of the 64 node 'Caltech Cosmic Cube'. The proposed algorithm is described below.

Initially, the vertices in the application's communication graph are assigned

randomly to nodes in the machine. Two vertices are then chosen at random and swapped. If this swap reduces the 'cost' of the placement (as defined later), the vertices are kept in their new locations. If the cost increases as a result of the swap, the vertices are moved back to their original locations with a probability dependent on the magnitude of the cost increase and the current 'temperature'.

The process of picking and swapping vertices is repeated many times while the temperature is gradually reduced. Initially the temperature is set to a large value causing most swaps to be accepted, regardless of their cost. As the algorithm proceeds, the temperature is reduced until only swaps resulting in cost reductions are accepted. By initially accepting detrimental swaps, the algorithm typically avoids becoming trapped in local minima. Towards the end of the algorithm's execution the solution is presumed to be close to the global minimum-cost solution and so few or no detrimental swaps are accepted.

The cost of a particular placement is defined by some expression approximating the quality of the placement with better quality placements having lower costs. In Steele's work, the cost of a placement is computed by finding the shortest path routes between all connected vertices using Dijkstra's algorithm. Rather than repeating this relatively expensive computation after each swap, Steele's implementation updated the previously calculated cost by recomputing only the routes connecting the swapped vertices.

The number of swaps to attempt and the rate at which the temperature is changed is dictated by an 'annealing schedule' and can have a major impact on the execution time of the algorithm and the quality of the placement solution. I describe one such schedule in detail later in this chapter when describing my own simulated annealing based application placement algorithm.

Towards the end of the 1980s, application placement appeared to be becoming less important as supercomputer network architectures improved, ap-

pearing to render computationally intensive placement algorithms unnecessary [Dal87]. In addition, network and problem sizes remained small: so small in fact that exact linear-programming based placement algorithms remained a meaningful point of comparison in application placement research [XH91]. In this environment, simpler sequential placement algorithms gained favour over more computationally expensive algorithms such as simulated annealing.

In contemporary application placement algorithms, 'low temperature' simulated annealing algorithms are occasionally employed as a post-processing step for other placement algorithms [HS11]. Other heuristics are used to generate an initial placement solution.

Recursive partitioning

As problem and machine sizes have grown and network utilisation has once again become an important factor in application performance [NPMA09], more complex optimisation algorithms have reappeared in the literature. One popular approach employs graph partitioning algorithms such as METIS [KK98] to perform recursive bipartitioning based placement [PSJ+14, HS11, PR96]. This placement process is illustrated in figure 6.5.

In the first step, the application communication graph and machine connectivity graph are each bipartitioned such that the number of edges cut by the partitions is minimised while the size of each partition remains similar. Each half of the communication graph is associated with one of the halves of the machine connectivity graph. The partitioning process is then repeated recursively, further partitioning the application and machine graphs. The process halts when either graph can no longer be partitioned. The vertices in the communication graph are then placed on their associated node in the connectivity graph.

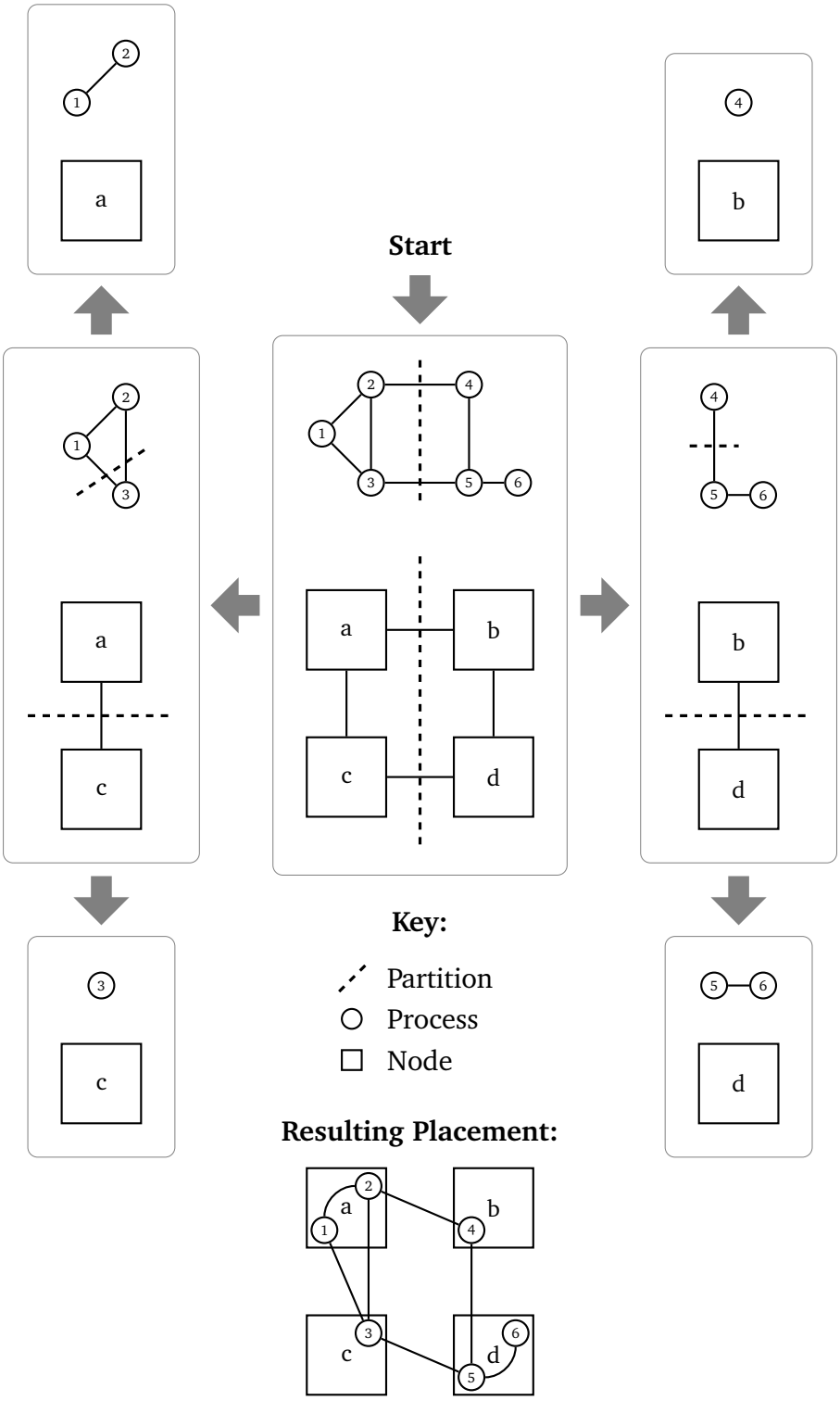


Figure 6.5: Illustration of application placement by recursive partitioning. Each node has the capacity to hold two vertices.

Unfortunately, recursive partitioning-based techniques are known to perform sub-optimally when the number of nodes is not a power of two [ST97]. In addition, placement problems are often subject to multiple orthogonal constraints. For example, a vertex in an application's communication graph may consume a certain number of processors and some quantity of SDRAM. Partitioning algorithms require a metric with which the size of each partition may be measured to ensure that the two halves are balanced. When multiple orthogonal resources exist, a single metric is insufficient to express this requirement. As a result, partitioning algorithms are of limited use in applications whose placement is subject to several orthogonal requirements, such as neural modelling software on SpiNNaker.

Topology-specific approaches

Other approaches employ other optimisation techniques that are specialised for use with applications and networks with specific connectivity patterns, for example, trees [JMT14, Trä02]. Unfortunately, these algorithms cater neither for SpiNNaker's hexagonal torus topology nor the range of communication topologies found in its target applications.

6.2.2 Circuit placement algorithms

The circuit placement problem in chip and FPGA design is similar to that of application placement: communicating components must be assigned positions in a chip so that wiring congestion is controlled.

Following the march of Moore's 'Law' [Moo75], the number of components in modern chips and FPGAs has been increasing at an exponential rate similar to that of core counts of supercomputers (see figure 6.6). Today, in 2016, the largest chips contain billions of transistors, compared with millions of pro-

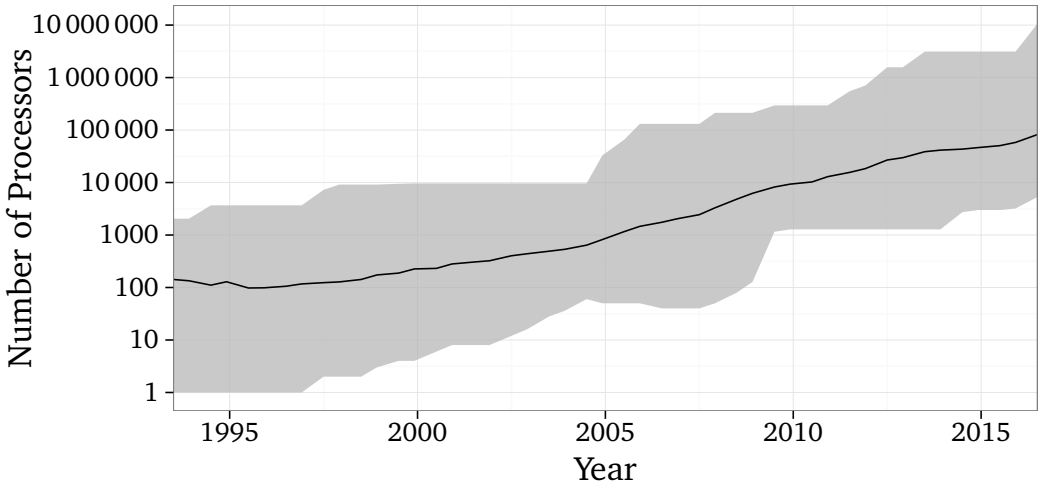


Figure 6.6: Historical processor (core) counts for the 'Top500' supercomputer installations. The line shows the mean number of processsors and grey ribbon indicates the range. Data reported by the 'Top500 List' [Meu16].

cessors found in supercomputers. In terms of the raw number of processors, the world's largest supercomputers are at a scale comparable with the number of components in chips designed fifteen years ago. The mean size of today's 'Top 500' supercomputers is somewhat smaller still, similar in scale to chips designed over 20 years ago.

In view of of the similarity of the chip and application placement problems and the relative lead the field of chip placement has in terms of placement size, it appears reasonable to look to chip placement to suggest future application placement techniques.

The popular approaches to circuit placement can be grouped into three broad categories [KLMH11]:

Partitioning-based As in application placement, the circuit is recursively partitioned to assign related components to areas of a chip.

Analytical These algorithms approximate the placement problem as a simpler problem to which exact solutions can be computed inexpensively.

Simulated annealing As in application placement, simulated annealing algorithms selectively apply random permutations to a placement causing placement quality to improve.

I briefly introduce each technique before assessing its suitability for use in application placement in SpiNNaker. The interested reader is encouraged to refer to 'VLSI physical design: from graph partitioning to timing closure' by Kahng *et al.* [KLMH11] for a more thorough introduction to the field of circuit placement and the techniques that I summarise here.

Partitioning-based placement

Partitioning based placement has a long history with early examples dating back to the 1970s [Bre77]. As in recursive partitioning based application placement algorithms, the input circuit is partitioned alongside the target topology to assign placements. Again, graph partitioning software such as METIS [KK98] is used to partition the circuits but since the target topology is the 2D surface of a chip, simple geometric partitioning is used to divide the surface of the chip. More recent partitioning based placers such as Capo [RPA+05] employ additional heuristics such as 'terminal propagation' where connections between adjacent partitions are considered to attempt to align components on either side of a cut.

Unfortunately, as with its use in application placement, these partitioning techniques are not ideal. Consequently, partitioning based placement techniques have largely been superseded in chip design [MHK15].

Analytical placement

The current generation of placement algorithms attempt to approximate placement problems as simpler problems with cheaply computed, closed-form solu-

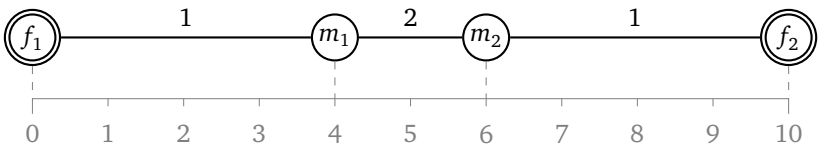


Figure 6.7: An optimal 1D quadratic placement solution for two fixed vertices (f_1 and f_2), two movable vertices (m_1 and m_2) and three weighted edges.

tions. One of the common approximations used by these approaches is modelling the components to be placed as point-like objects. As the number of components in digital circuits has grown, this approximation has also improved, resulting in analytical placement becoming the most popular approach in use today [MHK15].

One popular analytical technique is 'quadratic placement' [KLMH11, SSJ08]. In this algorithm the components in a circuit are approximated as infinitesimal objects with wires between them modelled as spring-like forces. Unlike physical springs, which obey Hooke's law, the force exerted by a connection is proportional to the *square* of the distance between its terminals.

To illustrate this approach, consider the simple one-dimensional placement problem presented in figure 6.7. In the quadratic placement technique, the cost of a placement is modelled as the weighted sum of the squares of the distances between connected components. In this case this yields the expression:

$$\text{Cost} = 1(f_1 - m_1)^2 + 2(m_1 - m_2)^2 + 1(m_2 - f_2)^2$$

Where f_1, f_2, m_1 and m_2 represent the positions of the four components. An optimal placement is one which minimises this function. To find values which minimise a quadratic, the equation is differentiated yielding a system of linear equations. This system of equations is then solved to find the positions of the movable vertices which minimise the quadratic cost function.

Quadratic placement is extended to two or more dimensions by simply applying the process for each dimension separately. Unfortunately, a straightforward extension to non-Euclidean geometries such as hexagonal toruses is non-trivial. For example, quadratic placement relies on cost increasing monotonically as two points are moved in opposite directions, a property which does not apply to torus topologies. This same problem also applies to the approximations used by other analytical techniques.

An additional challenge posed by quadratic placement is that some vertices' locations must be fixed to provide a 'spreading' force preventing degenerate solutions where all vertices are given the same coordinates. In circuit placement, the locations of 'pads' used to connect components in a circuit to the outside world are typically fixed and provide convenient anchors during placement. Algorithms such as SimPL [KLM12] employ a combination of analytical techniques to add virtual 'anchor' components which prevent degenerate solutions from forming. Unfortunately, this approach further adds to the difficulty of porting the technique to non-Euclidean geometries. These techniques also work better for larger circuit sizes, making analytical techniques non-ideal for today's relatively small (by chip placement standards) supercomputer placement problems.

Simulated annealing

Simulated annealing was originally developed specifically to be applied to the problem of circuit placement [KGV+83]. Throughout the late 1980s and early 1990s, simulated annealing was the dominant circuit placement technique when the first chips breaching one million components appeared [BR97, SSV85]. Due to the increased effectiveness of quadratic placement as problems grew beyond tens of millions of nodes, simulated annealing has fallen

out of fashion as a circuit placement technique. Despite this, various efforts continue to be made to enhance the simulated annealing process' performance by attempting to partition the input problem into several independent problems [CBZ10, HNCB00] and parallelise the algorithm [LBP08].

Simulated annealing notably remains popular as a stand-alone placement tool in niche applications where flexibility is important. For example, the flexible, multi-FPGA, Verilog-To-Routing (VTR) software suite [LGW+14] and Open Source Arachne-PNR [Cse16] use simulated annealing to produce placements within multiple FPGA architectures.

Since application placement problems are only now beginning to reach the scales at which simulated annealing became popular in circuit design, it is likely to be a good choice for application placement over the next few years. In addition, simulated annealing's flexibility potentially makes it a good choice to handle the wide variety of architectures in use by modern supercomputers.

6.3 Placement by simulated annealing

As we have seen, existing software placement techniques are not well suited to the large-scale applications projected for the SpiNNaker architecture. In the field of circuit placement, various algorithms have been proposed which are explicitly designed to cope with placement problems of similar or larger sizes. While analytical placement algorithms can produce high quality placement solutions for extremely large problems, the current – and near future – generations of application placement problems are still too small. Simulated annealing, however, has a proven record in circuit design at the scale of modern application placement problems. In addition, the flexibility of simulated annealing makes it easy to adapt to new network architectures and, in this

case, SpiNNaker's hexagonal torus topology.

To demonstrate the suitability of simulated annealing for modern applications, I have implemented a simple simulated annealing based placer for the SpiNNaker architecture. Compared with previous simulated annealing algorithms for application placement, this algorithm is intended to cope with much larger problem instances and draws inspiration from the techniques used by circuit placement algorithms.

In contrast with circuit placement algorithms built on simulated annealing, my implementation represents the problem differently to handle the nuances of application placement. In circuit placement, for example, exactly one component may be placed in a given space while in application placement, several applications may be placed on the same node (e.g. on different cores, or as separate processes). These differences, along with details of the cost functions and annealing schedule used are described in the remainder of this section.

6.3.1 Representation

A typical SpiNNaker application is made up of a number of processes which are executed by individual processor cores, consuming some quantity of on-chip resources such as shared SDRAM. These processes communicate via multicast packets which are sent via SpiNNaker's network.

In the proposed placement algorithm, an application is defined as a graph where the 'vertices' represent the processes as defined above and the multi-edges, referred to here as 'nets', represent directed flows of multicast packets from one vertex to several others. Each vertex consumes some quantity of on-chip resources (typically one core and a number of bytes of on-chip RAM). The placement algorithm must assign each vertex to a specific SpiNNaker chip (i.e. a node in SpiNNaker's network) such that the resources available on that chip

are not exhausted.

An additional soft constraint is imposed: the placement algorithm should attempt to place connected vertices as close together as possible. Nets may be annotated with a 'weight' which provides a hint to the placer indicating the relative importance of a particular net. A larger weight indicates the placer should try harder to keep the vertices it connects closer together. Because this constraint is soft, the placer should make a 'best effort' attempt to obey the constraint but is not obliged to achieve a perfect placement solution.

The SpiNNaker topology is represented as a 2D hexagonal torus topology with homogeneous inter-chip links. Each chip is defined to have a certain quantity of each resource consumed by vertices. For example, SpiNNaker chips nominally contain 17 application processors and 128 MB of on-chip SDRAM. Some chips, however, may have fewer working cores while others may be non-functional, thus effectively having no resources.

Additional constraints

Real-world applications may also apply some additional constraints on placement solutions. Two such constraints are considered here.

Some applications require that certain vertices are always placed on a specific chip, for example to enable communication with an external device attached directly to a SpiNNaker chip. The placement algorithm must obey this requirement, treating constrained vertices as a special case.

In other applications, in-memory communication may be required between certain vertices and therefore these vertices must always be placed on the same chip. This constraint may be implemented by preprocessing the application graph such that constrained vertices are merged into a single vertex whose resource requirements are the sum of the original vertices.

6.3.2 Algorithm description

As outlined earlier, simulated annealing swaps pairs of vertices keeping beneficial swaps as well as some proportion of detrimental swaps. In the remainder of this section I describe the exact processes by which vertices are selected and swapped. I also consider the choice of cost function for use with a hexagonal torus topology before specifying the annealing schedule used.

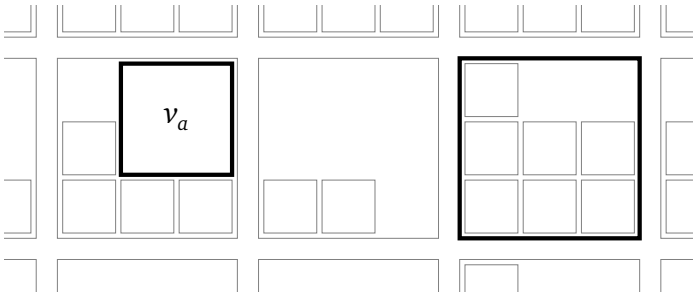
Generating candidate swaps

Simulated annealing based placement schemes move vertices by swapping random pairs of vertices. In application placement problems, where vertices may differ in resource requirements, it is not always possible to swap arbitrary pairs of vertices. For example, if a 'larger' vertex is swapped with a 'smaller' one it may not fit in the space left behind by the smaller vertex.

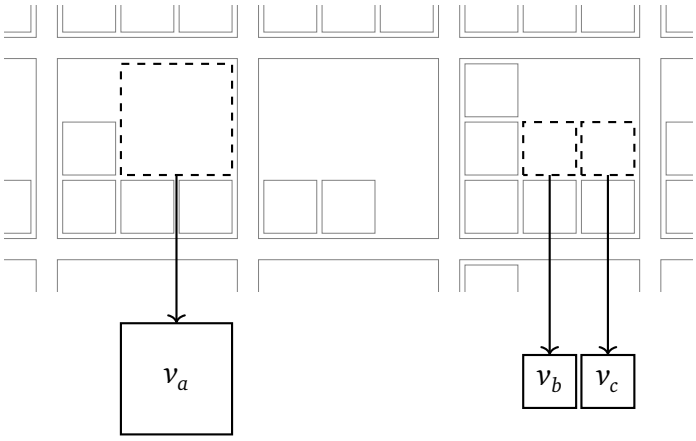
Rather than blindly picking pairs of vertices to swap and abandoning those which don't fit, the process described in figure 6.8 is used. This swapping process reduces the likelihood that swaps involving larger vertices are aborted when the vertices on the target chip are individually too small to free up enough room to complete a swap.

Cost function

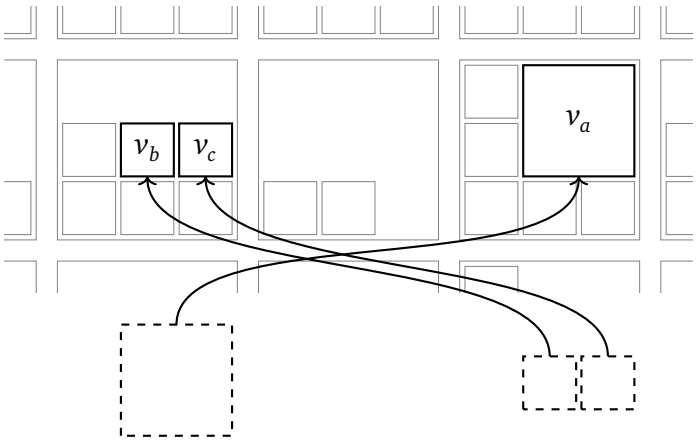
For each swap made by the algorithm, the change in the placement's 'cost' must also be determined. In principle, the ideal cost function should accurately predict network congestion by performing routing and modelling the resulting network traffic. This approach was used by early application placement algorithms [Ste85] but incurs a large performance overhead. Since the simulated annealing algorithm must re-evaluate the cost function before and after ev-



(a) A random vertex, v_a is selected along with a random target chip (right).



(b) Vertex v_a is removed from its chip. Vertices are removed from the target chip until enough space is available to fit v_a. In this example, vertices v_b and v_c are removed.



(c) Vertex v_a is swapped with vertices v_b and v_c if there is room. If there is not enough room the vertices are returned to their original locations and the swap is aborted.

Figure 6.8: Selecting vertices to swap.

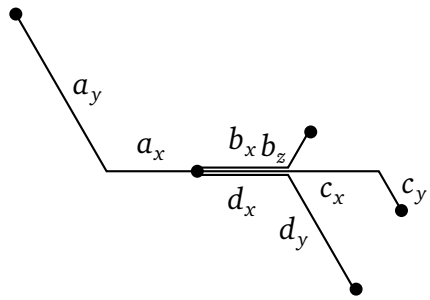
ery swap, a cost function which is slow to evaluate can result in unacceptable execution time for the algorithm as a whole.

A common alternative to measuring congestion directly used by many circuit placement algorithms is to approximate the network resources consumed. In the case of multicast nets, rather than computing efficient multicast routes, an estimate is made based on the relative locations of the vertices. Two popular techniques are described below [KLMH11, §4.2].

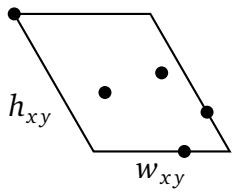
In the star model, the cost of a net is found by summing the distances from the net's source vertex to each sink vertex in turn as illustrated in figure 6.9a. In a hexagonal mesh and torus topology, the distance between two points can be cheaply determined (as described in chapter 4) making this cost function inexpensive to compute. This model will overestimate network resource usage for nets with more than two vertices since, in practice, multicast routing algorithms will tend to generate routes which share some hops.

In the Half-Perimeter Wire Length (HPWL) model, a bounding box is drawn around all of the vertices connected by the net and the cost of the net is estimated as half of the perimeter of this bounding box. In a non-hexagonal torus or mesh topology, the HPWL model exactly matches the length of an optimal multicast route for nets with two or three vertices. In nets with a greater number of vertices, HPWL underestimates the length of an optimal route by a factor of $\sqrt{n}$ for nets with n vertices [CH79].

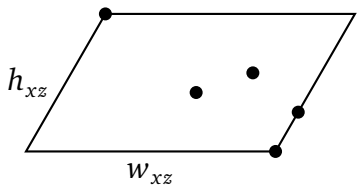
The three non-orthogonal axes of hexagonal mesh and torus networks mean that there are several ways in which a bounding box may be defined. Specifically, the bounding box may extend along any pair of axes resulting in three possible bounding boxes and, thus, three HPWL values for any net as illustrated in figures 6.9b, 6.9c and 6.9d. The relative sizes of these three bounding boxes varies depending on the specific arrangement of vertices in the net, meaning



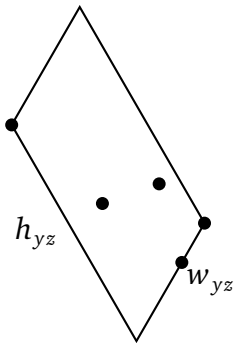
(a) Star $= a_x + a_y + b_x + b_z + c_x + c_y + d_x + d_y$



(b) $\text{HPWL}_{xy} = w_{xy} + h_{xy}$



(c) $\text{HPWL}_{xz} = w_{xz} + h_{xz}$



(d) $\text{HPWL}_{yz} = w_{yz} + h_{yz}$

Figure 6.9: Net cost estimation functions for hexagonal toruses and meshes.

that no single bounding box is appropriate for all situations.

A single HPWL-like metric for hexagonal toruses can be produced by taking the minimum of the half-perimeters of the three different bounding boxes defined for a hexagonal toruses. Like the HPWL function for non-hexagonal toruses, this 'hexagonal HPWL' cost function exactly predicts the length of optimal routes for all nets with two or three vertices.

An experiment was performed to determine which cost function provides the most accurate results. 50 000 nets connecting between two and thirty uniform randomly selected vertices in a 30×30 hexagonal torus topology were generated. The difference between each cost function and the 'true' cost of a route generated by the NER routing algorithm is plotted in figure 6.10.

These plots confirm that the cost is overestimated by the star model and un-

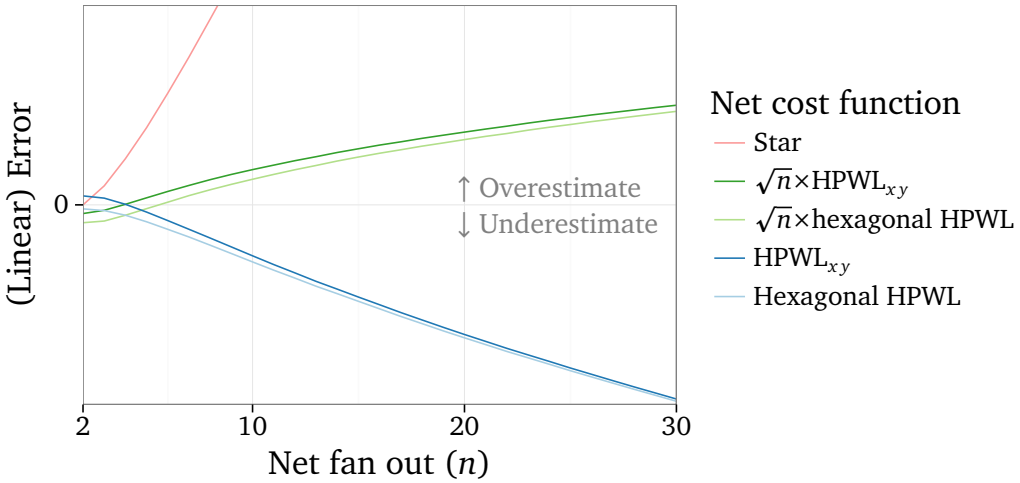


Figure 6.10: Estimation error of various cost functions compared with routing solutions. Positive and negative errors are equally bad. Y-axis limit clipped for clarity. The error of the star cost function grows linearly with fan out beyond the axis limit.

derestimated by both the hexagonal HPWL and HPWL$_{xy}$ models. The error for the star model grew more quickly than both HPWL variants. Both hexagonal HPWL and HPWL$_{xy}$ perform similarly though, as expected, hexagonal HPWL is more accurate for low fan out nets.

To attempt to compensate for the theoretical $\sqrt{n}$ error in the HPWL cost, the Hexagonal HPWL and HPWL$_{xy}$ scaled by $\sqrt{n}$ are also shown. While this correction does not eliminate the error of these functions, it does reduce it. This is probably because the error only tends towards $\sqrt{n}$ for very large n [CH79] and also because the error is being computed against a realistic, non-optimal routing algorithm.

On the basis of these results, the star cost function is rejected on the grounds that it produces the least accurate cost estimates. Though the $\sqrt{n}$-scaled hexagonal HPWL cost function produces the smallest error in general, its execution time is three times greater than $\sqrt{n}$-scaled HPWL$_{xy}$ whose accuracy is very similar. Since the execution speed of the cost function has a significant im-

pact on the execution time of the simulated annealing algorithm, I have used the $\sqrt{n}$-scaled HPWL_{xy} net cost function as a trade-off between accuracy and performance in this work.

Annealing schedule and acceptance function

The annealing schedule and acceptance function described below are taken directly from that used by the VPR FPGA placement software [BR97]. Due to a combination of time constraints and this schedule being known to work well, alternatives have not been considered in this thesis.

Starting with an initial random (but valid) placement, N swaps are performed to gather statistics about the problem, where N is the number of vertices in the graph. The initial annealing temperature, T, is set to 20 times the standard deviation of the cost changes produced by the N swaps.

The annealing algorithm now performs a round of $E \times N^{1.33}$ random swaps where E is a tunable parameter controlling 'effort'. In all experiments reported in this thesis, E is set to 1. Increasing E roughly linearly increases execution time for diminishing returns in placement quality improvements. Each random swap is accepted with a probability:

$$P_{\text{accept}} = \begin{cases} 1 & \text{if } \Delta C \leq 0 \\ e^{-\Delta C/T} & \text{otherwise} \end{cases}$$

Where ΔC is the change in cost resulting from the swap and e is Euler's number. This probability is 1 for any beneficial or neutral change. Small cost increases are accepted with a greater probability than large cost increases. At higher temperatures, larger cost increases are accepted with a greater probability than at lower temperatures.

After a round of $E \times N^{1.33}$ of swaps has been performed, the temperature is updated according to $T_{\text{new}} = \alpha T_{\text{old}}$ where α is determined thus:

$$\alpha = \begin{cases} 0.50 & \text{if } R_{\text{accept}} > 0.96 \\ 0.90 & \text{if } 0.80 < R_{\text{accept}} \leq 0.96 \\ 0.95 & \text{if } 0.15 < R_{\text{accept}} \leq 0.80 \\ 0.80 & \text{if } R_{\text{accept}} \leq 0.15 \end{cases}$$

Where R_{accept} is the proportion of swaps which were accepted by the algorithm in the previous round. This approach attempts to maximise the time spent at temperatures in which some, but not all, swaps are accepted since this is claimed to be when the greatest improvements to placement quality are made [BR97].

The algorithm terminates when $T < {}^{0.005C}/_{N_{\text{nets}}}$ where C is the current cost of the placement solution and N_{nets} is the number of nets in the placement problem.

Towards the end of placement, the solution is likely to be close to the (ideally global) minimum cost. In already-good solutions it is likely that swapping vertices between nodes which are far apart in the network will result in large cost increases. As a result, towards the end of the anneal, swaps are only considered over a limited distance, D_{limit}. By limiting the distance over which swaps occur, the proportion of swaps which are likely to be accepted is also increased. D_{limit} is initially set such that all possible swaps are considered. It is then updated according to the following formula at the end of each round of swaps:

$$D_{\text{limit}}^{\text{new}} = \max(1, \quad D_{\text{limit}}^{\text{old}}(1 - 0.44 + R_{\text{accept}}))$$

This rule attempts to limit swap distance such that the proportion of accepted

swaps does not drop below 0.44 for as long as possible, that is, such that some, but not all, swaps are accepted for as long as possible.

6.4 Evaluation

A good placement algorithm should produce placements which result in efficient use of network resources. In addition, in SpiNNaker, routing table usage is an important concern since routing table entries are a finite resource. Finally, the execution time of the placement algorithm itself is relevant since, especially in applications where experiments are of short duration, placement execution time should not dominate application execution time.

In this evaluation, the simulated annealing based placement algorithm introduced in this chapter is compared with a number of baseline algorithms representative of existing application placement techniques. I consider a combination of existing SpiNNaker applications and larger-scale synthetic benchmarks representative of possible future applications. I compare the performance of the placement algorithms under consideration using a combination of software models and confirm these results using experiments running on SpiNNaker hardware.

6.4.1 Baseline algorithms

To provide a baseline against which to compare my simulated annealing based placement algorithm, three other placement algorithms are considered:

Hilbert The sequential placement algorithm used by existing SpiNNaker software and similar to the algorithm provided by Cray's extensions to SLURM. This places vertices along a Hilbert curve path in breadth-first traversal order of the application communication graph.

RCM The 'graph similarity based mapping' algorithm described by Hoefler and Snir [HS11]. This assigns vertices and fills chips in Reverse Cuthill McKee order [CM69].

Random A random placement algorithm which places vertices uniform-randomly with no regard for connectivity. This algorithm serves as an experimental control.

6.4.2 Existing applications

To verify the suitability of simulated annealing based placement for existing SpiNNaker software, a representative selection of benchmark applications has been collected. The communication graphs for these benchmark applications are then used to compare the performance of each placement algorithm against a set of simple performance metrics. In addition, a subset of the applications were modified to use each of the routing algorithms under test and changes in their performance are reported.

Table 6.1 summarises each of the benchmarks. While these applications are much smaller than those expected to run on new, large-scale SpiNNaker systems, these benchmarks provide a meaningful indicator of the types of connectivity which might be present in future networks. Further experiments using synthetic benchmarks, described later, will extend this study to applications at the scale SpiNNaker is designed to operate at.

Communication graph placement performance

Each benchmark's connectivity graph was placed on into an idealised 13×13 hexagonal torus topology and then routed using the NER algorithm. This process was repeated 1000 times with various performance metrics recorded

Name	Vertices	Nets	Fan out	Notes
Microcircuit	1338	760	94.50 (98)	Spiking neural model of a cortical microcircuit [PD14] implemented in 'PyNN SpiNNaker' [KTK+16].
Sudoku	299	109	15.86 (21)	Spiking neural model which solves Sudoku puzzles implemented in 'PyNN SpiNNaker' [KTK+16].
Card-Sorting	469	919	1.06 (3)	Spiking neural model which solves the Wisconsin card sorting test [Aub15] implemented in 'Nengo SpiNNaker' [MKSF15].
CConv	2560	12 020	3.48 (17)	Spiking neural model which performs 512D circular convolution [Eli13] implemented in 'Nengo SpiNNaker' [MKSF15]. A key component in the Spaun neural model [ESC+12].
Parse	855	3046	3.26 (64)	Spiking neural model which performs simple sentence parsing implemented in 'Nengo SpiNNaker' [MKSF15].
MU0	1084	1084	1.38 (3)	A digital circuit simulation of the data path of a simple computer processor [Pau16] implemented using the digital circuit simulator example program included with 'Rig' [HMK15].

Table 6.1: SpiNNaker application benchmarks. 'Fan out' refers to the fan out of the nets in the communication graph and not necessarily the total fan outs of individual vertices. Both the mean and (maximum) fan out are shown.

which are analysed below. Experiments were spread across a cluster of idle workstations with 3.10 GHz Intel Core-i5-2400 CPUs and 8 GB of RAM.

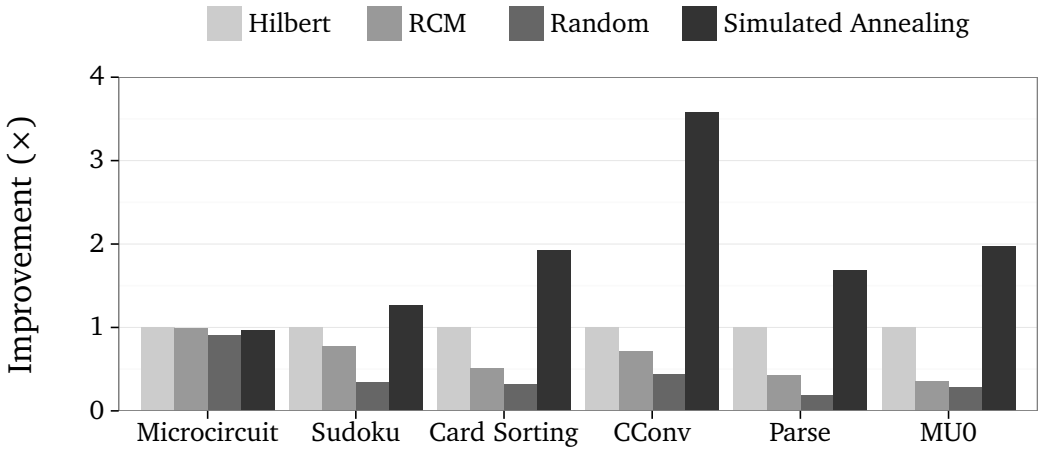
To measure the 'quality' of the placement solutions generated, the total number of hops taken by the generated routes is compared in figure 6.11a. A better placement solution should require fewer hops and, as an indirect result, achieve lower network congestion. Since the hop counts vary greatly between benchmarks, these results are normalised against the number of hops produced by the Hilbert placer, the placement algorithm used by default in pre-existing SpiNNaker software.

In the majority of benchmarks, the simulated annealing based placement algorithm achieves the greatest reduction in hop counts. The RCM and Random placement algorithms both consistently produce results similar to or worse than those of the Hilbert placement algorithm.

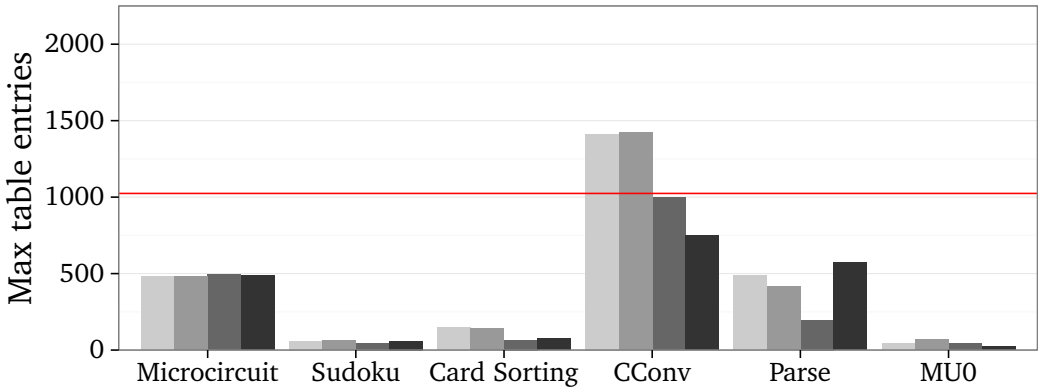
Unlike the other benchmarks, the 'Microcircuit' benchmark notably shows little difference in placement quality amongst the four algorithms. This may be explained by the fact that this network is largely all-to-all connected meaning that a placement function can do little more than place the vertices tightly together to improve quality.

The second metric considered is the number of routing table entries required in the worst case after placement and is shown in figure 6.11b. As is the practice of the applications modelled by these benchmarks, routing tables are compressed by the 'Ordered Covering' algorithm [MHG16] and default routing is used when possible.

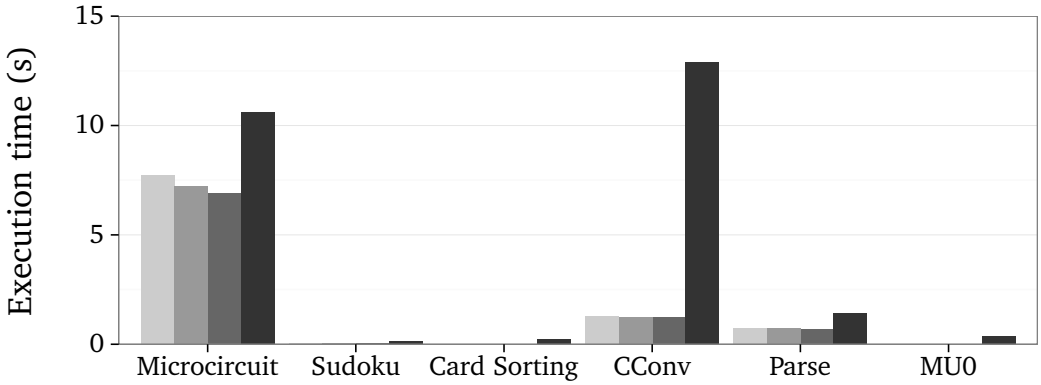
Though many of the benchmarks do not run out of routing entries when placed with any of the algorithms, the 'CConv' network notably runs out of table entries when placed by all but the simulated annealing and random placement algorithms. This can be attributed to the large number of nets the 'CConv'



(a) Placement quality (routed net length reduction vs. Hilbert placer)



(b) Maximum routing table entries (red line at 1024 entries)



(c) Placement algorithm execution time

Figure 6.11: Application benchmark placement performance.

network uses by comparison with other models.

The random placement algorithm can often achieve competitive routing table sizes because it tends to spread out vertices over the entire machine. Because longer routes don't always result in more routing entries being required (thanks to default routing), this also results in the routing entries being spread more thinly across the whole system. In these experiments the majority of benchmarks do not fill the 13×13 hexagonal torus topology supplied, meaning that some networks can become spread out by the random placer. Though routing table usage drops, network utilisation, and potentially congestion, increases.

The third metric considered is the execution time of the placement algorithm, shown in figure 6.11c. The simulated annealing algorithm requires much longer execution times than the baseline placement algorithms considered; in several cases an order of magnitude longer. This is unsurprising since the baseline algorithms are comparatively simplistic greedy algorithms. In all applications considered, however, the execution time of the simulated annealing placement algorithm is multiple orders of magnitude shorter than other processes involved in loading executing the simulation on SpiNNaker.

The 'Microcircuit' benchmark is notable in that all of the placement algorithms take a long time to execute compared with the other benchmarks. This longer execution time can be attributed to a preprocessing step shared by every placement algorithm's implementation which handles 'same chip' constraints. Unlike the other application graphs, the 'Microcircuit' communication graph makes liberal use of these constraints and, due to the naïve implementation of the preprocessing step, all of the placement algorithms are slowed down by a constant amount.

Impact of placement on 'PyNN SpiNNaker'

The 'PyNN SpiNNaker' based neural models behind the 'Microcircuit' and 'Sudoku' benchmarks are compute bound and produce relatively small quantities of network traffic. Configuring 'PyNN SpiNNaker' to use each of the four placement algorithms under consideration did not result in any changes in application performance.

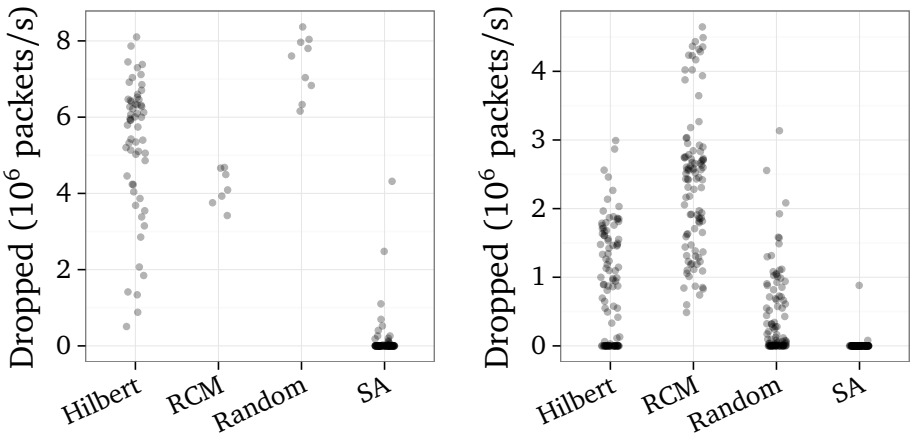
It is anticipated that future models built on PyNN SpiNNaker will grow in size and, unlike the Microcircuit model, not feature all-to-all connectivity and therefore begin to benefit from good quality placement.

Impact of placement on 'Nengo SpiNNaker'

The 'Nengo SpiNNaker' based neural simulations behind the 'CConv' and 'Parse' benchmark were run 100 times using each of the placement algorithms under test. For each run, the number of packets dropped by SpiNNaker during the application's execution was recorded and the results are presented in figure 6.12. Because the traffic produced by these simulations is multicast, it is not possible to sensibly normalise these packet counts against the rate of packets being produced[2] and so absolute numbers are given. Unfortunately the 'Card Sorting' neural model was not available for comparison in these experiments.

70% of simulations of the 'CConv' neural model resulted in fewer than 100 packets being dropped per second when placed by simulated annealing and 41% of runs dropped no packets at all. In 95% of runs SpiNNaker's routing tables were not exhausted. In the 'Parse' benchmark, 93% of simulations did not drop any packets when placed by simulated annealing and every simulation run's routing entries fit into SpiNNaker's routing tables.

[2]For example, one packet may be dropped in several places on its way to its many destinations.



(a) CConv (b) Parse

Figure 6.12: Rates of packet dropping in Nengo SpiNNaker simulations with different placements. Results shown jittered in the 'X' direction for clarity.

In both neural models, placement by the baseline placement algorithms resulted in fatal quantities of packets being dropped in most runs, rendering the baseline placement algorithms unsuitable for use with these models. In addition, for the 'CConv' model only 58%, 7% and 9% of placement solutions resulted in routes which fit in SpiNNaker's routing tables, for the Hilbert, RCM and Random placers respectively.

In these experiments, the simulated annealing algorithm reliably produces the best quality placements, often making the difference between simulations working or not. Unfortunately, the results show that the algorithm will, on rare occasions, produce unusable placements in terms of either network congestion or routing table use. A possible explanation for this behaviour is explored in §7.2.3 and possible solutions discussed. The experiments performed make use of irregularly-shaped hexagonal mesh topologies cut from larger SpiNNaker machines. The irregularities in the edges of the topology are hypothesised to be producing the same negative effect on route quality as high rates of HSS link faults noted in §5.3. Simulated annealing occasionally places some parts

of the network around obstructions at the edges of the network which may explain these infrequent failures.

Impact of placement on the 'Circuit Simulator'

The 'MU0' CPU simulation built using the 'circuit simulator' SpiNNaker application produces very little traffic in its default mode of operation and functions correctly when placed by all four placement algorithms.

The circuit simulator may be configured to run at higher speeds, directly increasing the density of network traffic generated. To judge the relative quality of all four placement algorithms the simulation was executed at a range of different speeds. By determining when or if the network becomes saturated, the relative placement quality may be compared.

When placed by the simulated annealing algorithm, at 400× the default simulation speed, the simulator becomes CPU bound but SpiNNaker's network does not become saturated or drop packets. By contrast, when placed by the baseline algorithms, the application saturates the network and drops packets before becoming CPU bound. Figure 6.13 shows the distribution of simulation speeds achieved for 100 runs of the simulator placed by each baseline placer. As in many of the other benchmarks, the Hilbert placer performs better than the Random and RCM placers with the RCM placer producing the placements which induced the greatest congestion.

6.4.3 Scalability

Unfortunately, due to the limited size of existing SpiNNaker applications it is necessary to turn to a synthetic benchmark to demonstrate the scaling properties of the simulated annealing placement algorithm.

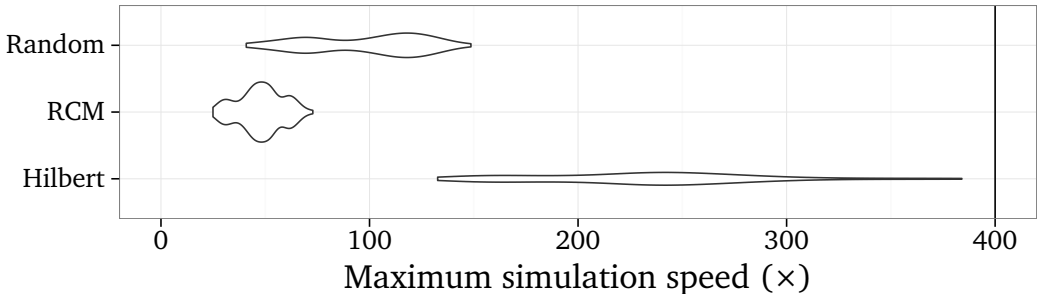


Figure 6.13: Simulation speeds at which the circuit simulator begins to drop packets.

A synthetic application graph is considered consisting of a 2D grid of vertices in which each vertex is connected with its neighbours according to a 2D Gaussian probability distribution. This application graph is then placed into a hexagonal mesh topology whose size and aspect ratio matches the application graph. This placement problem may be scaled up by increasing the number of vertices or by increasing the number of connections made by each vertex. The graph also has a 'natural' manual placement solution which is used as a gold standard in these experiments.

Placement quality is judged by comparing the total number of hops required to route all nets in the placed application graph against the number of hops needed when the vertices are placed 'manually' using their natural placement. Better placements should require a similar number of hops to the manual placement while worse placements will require more, implying greater network congestion. Worst-case routing table usage is calculated based on the use of routes which exploit default routing.

For each combination of placement algorithm and graph size, the experiment was repeated 20 times and the aggregated results are presented below. All experiments were spread across a cluster of idle workstations with 3.10 GHz Intel Core-i5-2400 CPUs and 8 GB of RAM.

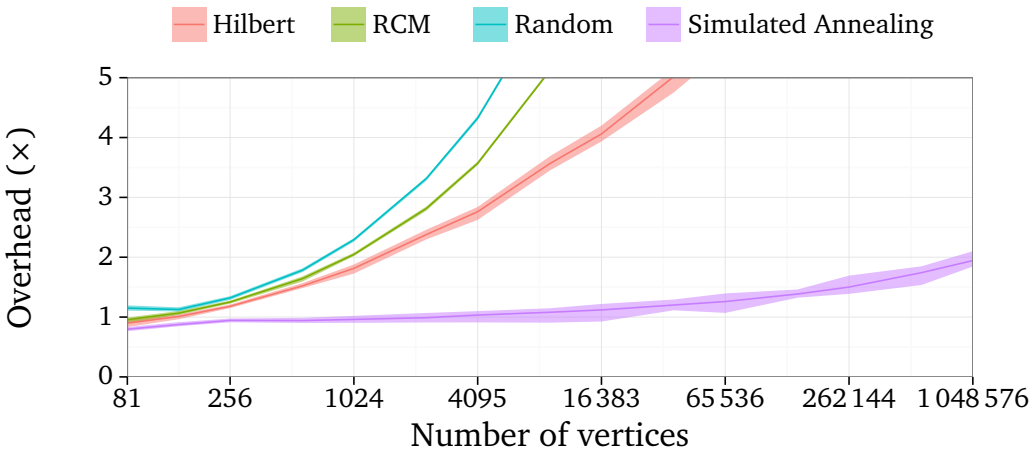
Graph size

This experiment intends to determine how the execution time and placement quality of each algorithm changes as the size of the application graph is increased. In this experiment, each vertex is connected to four randomly picked neighbours whose distribution of distances have a standard deviation of three hops. The results are shown in figure 6.14.

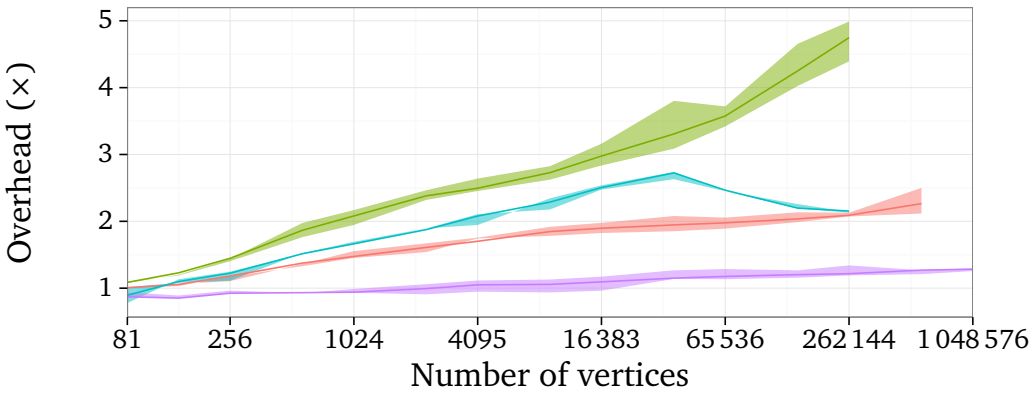
In figure 6.14a we can see that, as the placement problem size grows, existing placement techniques produce solutions requiring many more hops to route than a 'manual' solution. While the quality of the simulated annealing approach also falls for larger graphs, at 1 048 576 vertices, approximately the number in the largest planned SpiNNaker machine, only twice as many hops are required to route the placement compared with an ideal placement.

Beyond around a quarter of a million vertices, the quality of the placements produced by the baseline placement algorithms has dropped significantly. Due to the increased length of the routes required, routing these placed graphs requires more memory than was available in the cluster machines and so no results are reported.

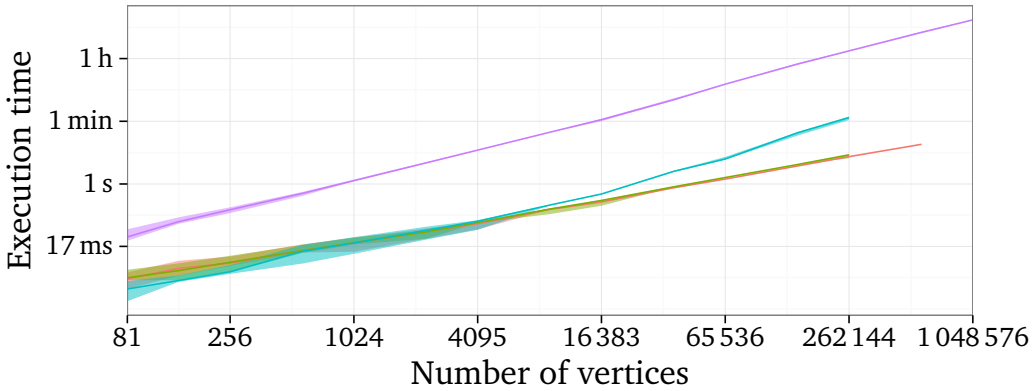
Figure 6.14c shows the execution time of the routing algorithms for each problem size. Once again, the simulated annealing algorithm's execution time consistently dwarfs the other algorithms but grows approximately linearly with the problem size. For the largest graph in the experiment, the execution time of the annealing based placer reaches approximately 12 hours. While such long execution times may not be ideal, for large placement problems, simulated annealing produces placements with substantially greater quality than the alternatives.



(a) Placement overhead (vs. manual placement)

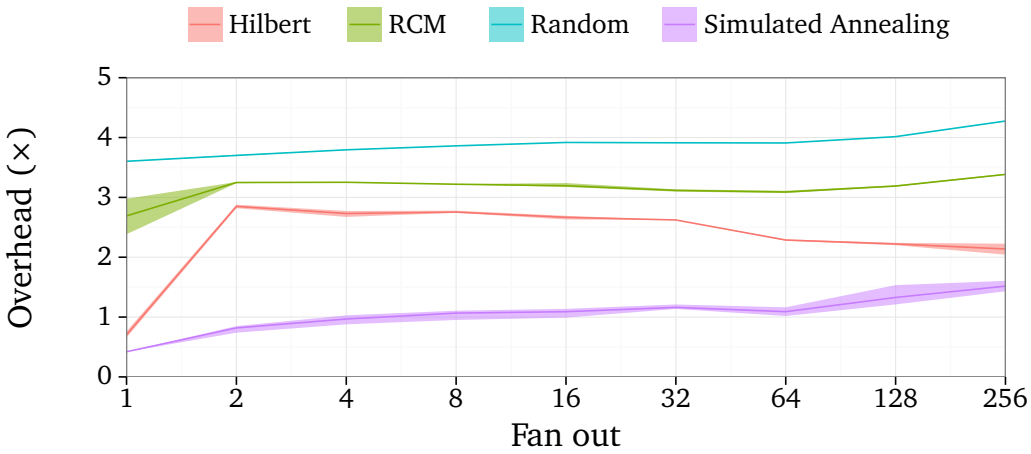


(b) Maximum routing table size

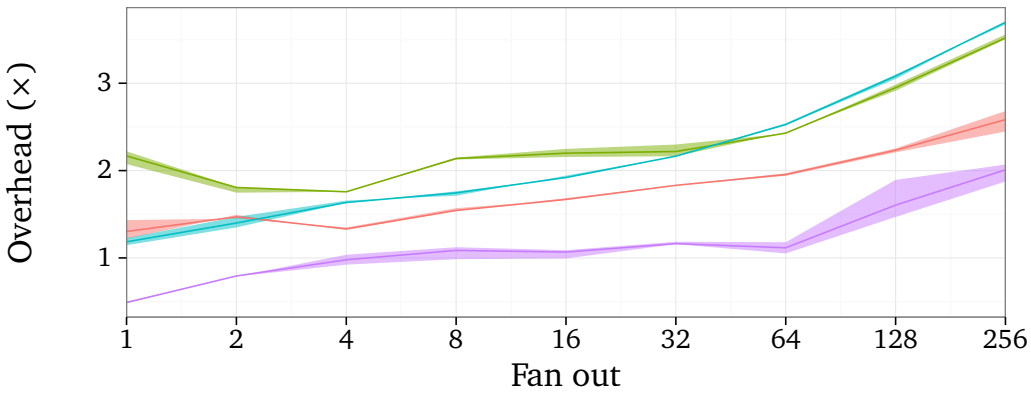


(c) Placement algorithm execution time

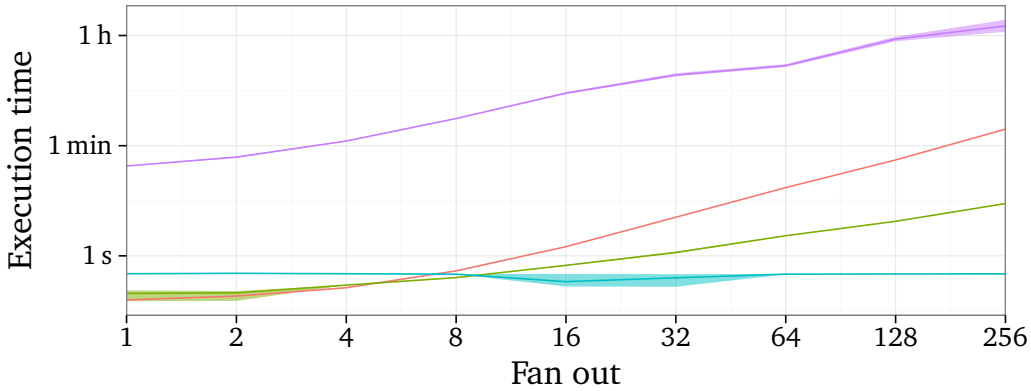
Figure 6.14: Scalability with respect to graph size. Lines give mean values, bands give range of results. Some results are absent due to out-of-memory errors during routing following placement of some larger graphs.



(a) Placement overhead (vs. manual placement)



(b) Maximum routing table size



(c) Placement algorithm execution time

Figure 6.15: Scalability with respect to fan out. Lines give mean values, bands give range of results.

Fan out

Another factor which can have a significant impact on placement execution time is the fan out of the nets in an application graph. A second experiment was conducted to determine the effects of increasing the fan out of the synthetic application graph. The number of vertices in the graph was fixed at 16 384 and the fan out of the nets connecting these vertices increased. Again, each combination of placement algorithm and fan out was tested twenty times and the results are shown in figure 6.15.

As shown in figure 6.15a, the quality of the placements produced does not change dramatically as the fan out is increased with the exception of the Hilbert placer which appears to perform better for very small and very large fan outs. In all cases however, simulated annealing maintains its lead in placement quality.

In all but the random placer (which does not consider connectivity), the placement algorithms' execution times grow in proportion to the fan out (figure 6.15a). Once again, simulated annealing is significantly slower than the other algorithms but scales approximately linearly with problem size.

6.5 Conclusions

In this chapter, I re-evaluated the suitability of simulated annealing for application placement in supercomputers. In the field of chip placement, simulated annealing has proven highly successful for placement problems similar in scale to application placement problems in modern supercomputers. Following this observation, I developed a simulated annealing based application placement algorithm for SpiNNaker's hexagonal torus topology based on techniques used by circuit placement algorithms.

Existing applications containing thousands of vertices have demonstrated improved performance when placed by simulated annealing. Included in these benchmarks are neural simulations built with Nengo SpiNNaker which do not function at all when placed by contemporary placement algorithms but run reliably when placed by simulated annealing.

Synthetic benchmarks have demonstrated that the proof of concept simulated annealing based placement algorithm developed as part of this work is able to maintain good placement quality in networks containing over one million vertices. Though execution times are considerably longer than existing application placement algorithms, taking over twelve hours for one million vertex networks, at these scales simulated annealing is the only placement algorithm able to deliver results within an order of magnitude of an ideal placement solution.

Of the baseline placement algorithms considered, the Hilbert placer consistently outperforms the other baseline experiments, followed by the RCM algorithm which often performed similarly. It is worth highlighting that the Hilbert algorithm exploits knowledge of SpiNNaker's network while the RCM algorithm is topology agnostic, possibly explaining the discrepancy in performance. The random placement algorithm consistently produced the worst placement quality confirming that even the simple baseline algorithms considered generate some benefit.

These results support the notion that circuit placement techniques may provide valuable insights for the development of future application placement techniques. As application placement problems continue to grow, adapting other placement techniques developed for circuit placement may become more challenging. A key difference between the circuit and application placement problems is that, while supercomputer network topologies come in many shapes

and forms, chips are largely 2D. Because of this, modern circuit placement techniques increasingly exploit geometric properties of the circuit placement problem which may be more difficult to adapt to supercomputer network topologies. However, at their current rate of growth, placement problems in supercomputers are likely to remain tractable to simulated annealing based placement techniques for a number of years to come.

Chapter 7

Conclusions and future research

The SpiNNaker architecture was designed to tackle the challenges presented by the simulation of biologically realistic neural networks. One of its distinguishing features is its network architecture which employs both an unconventional network topology and multicast router architecture. The hexagonal torus topology used by SpiNNaker was chosen to enable greater performance while maintaining ease of construction and scalability compared with conventional network topologies. SpiNNaker's router design centres around packets mimicking the neural 'spike' signals they are designed to convey by being small, multicast and not guaranteed to arrive at their destination. This novel design, though largely complete before this work began, left a number of open problems which this thesis has attempted to address.

In this concluding chapter I begin by summarising the answers to the research questions raised in chapter 1. This is followed by a discussion of new research topics which have been uncovered by this work.

7.1 Answers to research questions

Each of the three research questions are answered below.

1. Can the hexagonal torus topology be deployed and used in real, large-scale systems?

In chapter 3, I introduced a cabling scheme and assembly technique which has been used successfully to build a prototype SpiNNaker system with over half a million processor cores using the hexagonal torus topology. The techniques shown are expected to enable a final SpiNNaker machine of double this size to be built, filling a six metre long row of machine room cabinets.

Though SpiNNaker's processor count places it amongst some of the world's largest supercomputers (see figure 6.6, page 119), it is comparatively compact, filling one row of cabinets compared with commercial, warehouse-scale installations. In spite of this, the techniques described allow hexagonal torus topologies to scale arbitrarily without needing cables to span the machine.

Chapter 4 described an efficient and general technique for finding and enumerating shortest path vectors in hexagonal torus topologies. These developments bring the hexagonal torus topology in line with other topologies by enabling routing algorithms to exploit all possible paths in a network. Furthermore, chapter 6 demonstrated that placement algorithms are also adaptable to hexagonal torus topologies thanks to their similarity to 2D toruses.

Although, as this thesis highlights, hexagonal toruses lack many of the intuitive properties enjoyed by other topologies, it is still possible to reason about them with only limited computational effort. Now that the practicality and scalability of the topology has also been demonstrated in practice, it represents a credible option for future network architectures.

2. Does SpiNNaker's router architecture help, or hinder fault tolerance?

SpiNNaker's unconventional use of packet dropping to avoid deadlocks greatly simplifies the router architecture, part of the motivation for this design. In chapter 5, this feature is used to the advantage of PGS repair to add fault tolerance to existing routing algorithms. Compared with the often complex and wasteful methods used to tolerate faults in other networks, PGS repair incurs very little performance overhead in the presence of static faults.

Routing table usage does increase in the presence of faults, however, which may be a concern for applications which already require many routing table entries. Routing table usage, as well as other overheads, were most significantly increased in the presence of contiguous groups of network faults. This is because the PGS repair algorithm produces routes which pass tightly around the corners of faults, resulting in concentrations of routing table entries in those areas. Though the symptoms of this problem can be attributed to the design of SpiNNaker's multicast routing mechanism, the responsibility lies with the behaviour of the PGS repair algorithm. Potential improvements to the PGS repair algorithm are discussed later in §7.2.3.

The overall answer to this research question, therefore, is that the flexibility provided to routing algorithms in SpiNNaker's architecture is of great benefit, enabling arbitrary fault patterns to be inexpensively avoided.

3. How can the parts of a neural simulation be placed onto a large hexagonal torus topology to reduce network load?

In chapter 6, I explored a number of contemporary approaches to the problem of placing applications with irregular communication patterns into network topologies. I observed that researchers working on circuit placement for chips and FPGAs are tackling similar problems and working at scales as large as, or

larger than, those faced in application placement. Based on this I developed a simulated annealing based placement algorithm inspired by the techniques used in circuit placement, with specific adaptations for use in application placement and SpiNNaker's network topology.

The simulated annealing based placement algorithm consistently outperforms pre-existing placement algorithms included in benchmarks in terms of placement quality. In the case of one benchmark, simulated annealing based placement made it possible to run that neural simulation in real-time for the first time. At larger scales, simulated annealing was also found to be able to produce good quality placements in benchmarks containing over one million processes – the largest size supported by the SpiNNaker architecture.

The major shortcoming of simulated annealing based placement is its execution speed. Though its execution time grows in proportion to the size of the problem, the implementation used took over 12 hours to place a synthetic problem for the largest planned SpiNNaker machine. Though tractable, particularly given the relative output quality compared with the prior state of the art, the algorithm is unlikely to function comfortably as-is on larger problems.

The conclusion to be drawn from this result, however, is not just that simulated annealing is a good solution for today's placement problems but that circuit placement techniques in general could be successfully adapted to fulfil this role. The placement problems faced by chip designers are growing at roughly the same exponential rate as the size of supercomputers but circuit designs hold the lead in terms of problem size. Consequently, as approaches are retired by chip placement researchers, they may find new life in the field of application placement.

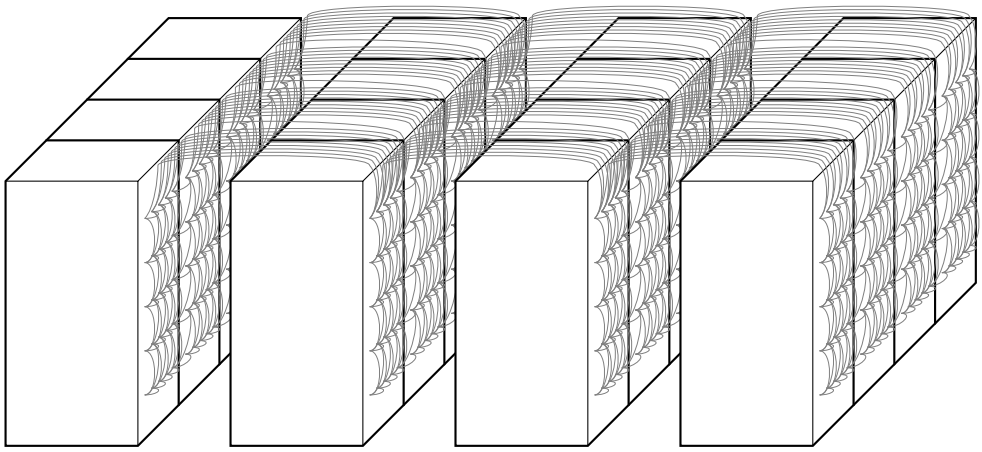


Figure 7.1: Multiple rows of interconnected cabinets.

7.2 Future research

Though the goals of this study have largely been met, there also remain some important limitations which future work may hope to address. Furthermore, this work has uncovered a number of new research areas warranting future enquiry. This section outlines a number of future lines of research.

7.2.1 Warehouse-scale cabling

In chapter 3 I developed and implemented a number of cabling schemes for the SpiNNaker architecture spanning up to a six metre row of machine room cabinets – a relatively small installation by current standards. In SpiNNaker, the cabling exists in a 2D plane (i.e. across the faces of the cabinets) but as the system is scaled up, a single row of cabinets will tend towards a 1D line. Since embedding a 2D structure in a 1D space necessarily results in long connections, this cannot scale indefinitely.

In conventional large-scale supercomputer installations, nodes are installed in rows of cabinets as illustrated in figure 7.1. From a 'bird's-eye' view, the

system approximates a 2D space, spread across the floor of a machine room. Therefore, in principle, the folding and interleaving techniques described in chapter 3 still apply. Unfortunately for SpiNNaker, cables connecting between rows of cabinets would be longer than the one metre limit imposed by its hardware because of the spacing between rows of cabinets. Future SpiNNaker systems will need to consider alternative link technologies. For example, a hybrid system could be used in which intra-cabinet connections continue to use the current HSS link technology while inter-cabinet links might use optical connections. This type of architecture could be supported by the use of pluggable 'SFP+' transceiver modules [Com01].

7.2.2 Cabling assistance for other architectures

A secondary result of the construction of prototype SpiNNaker systems in chapter 3 was the use of real-time guidance and feedback to assist cable installation. I am not aware of this technique's use by existing architectures and, following the success experienced in this project, it is possible that the technique may also be useful in conventional systems.

During the construction of prototype SpiNNaker machines, each cable took seconds to install compared with the minutes reported for existing systems [MYM11]. Part of this increase in efficiency appears to result from the immediate identification of mistakes made during cabling, saving time-consuming backtracking later on.

In many real-world network installations, units are less densely packed than in SpiNNaker and so longer cables are often required. As a consequence, cabling errors may become more likely as cabling patterns are spread over a larger area making them more difficult to visually verify. Like SpiNNaker, conventional networking hardware is often equipped with a generous range of

indicator LEDs and diagnostic facilities which might be used to implement real-time installation guidance. Future work could explore the use of this technique in the construction of other large-scale networks, such as data centres.

7.2.3 Congestion mitigation

In chapter 5 I found that contiguous network faults cause hot-spots of congestion and routing table depletion where the PGS repair algorithm routed many paths around the edges of faults. However, it is not just faults which can cause contiguous blockages in the network topology. In reality, researchers do not always require a full-sized SpiNNaker system to perform their experiments so large SpiNNaker systems are soft-partitioned on demand into many smaller machines [Hea16b]. To ensure isolation between partitioned sub-machines, HSS links between boards in different partitions are disabled. Because of SpiNNaker's 'wrapped triple' partitioning scheme, the resulting sub-machines have hexagonal *mesh* topologies (i.e. without wrap-around links) with irregular boundaries as in figure 7.2.

In partitioned systems, the 'tooth'-like gaps on the periphery of the network result in similar congestion to the HSS link failures considered in chapter 5. When a route is generated between nodes on opposite sides of a gap, the PGS repair process will produce a shortest-path route around it. Since many routes may be blocked by a single gap, a hot-spot may develop around the corners of the gap.

In chapter 6, the 'CConv' benchmark application was found to run correctly the majority of the time when placed by the simulated annealing algorithm but would occasionally fail by a significant margin. Preliminary experiments suggest these occasional failures are caused by placement solutions which place heavily communicating parts of the application on opposite sides of gaps along

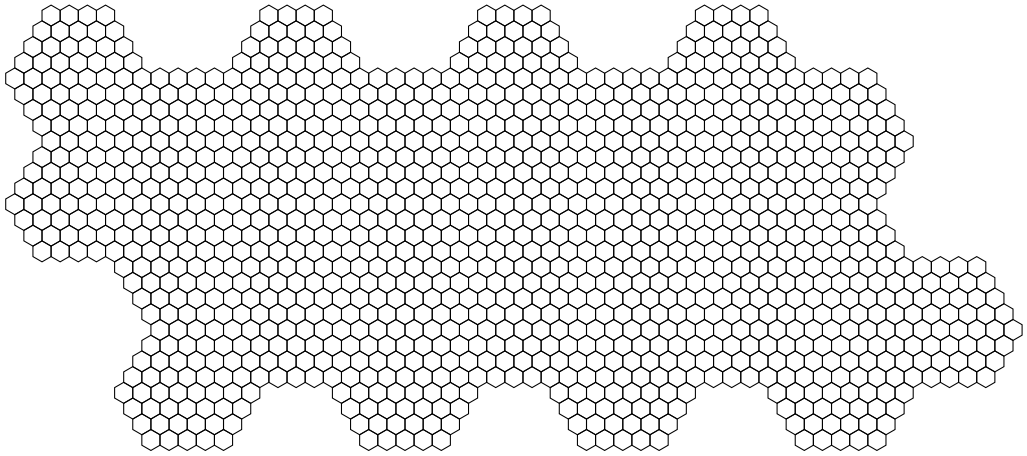


Figure 7.2: Irregular edges in a SpiNNaker system comprising 24 boards partitioned from a larger machine. Each hexagon represents a SpiNNaker chip. No wrap-around connections are present.

the perimeter of the network. Two possible approaches which future work may consider are presented below.

Avoiding hotspots with PGS repair

Network congestion around faults and network irregularities could be reduced by forcing the PGS repair process to take more varied routes around faults. For example, in circuit routing algorithms, routers avoid congestion by increasing the cost of routes which pass through congested areas [KLMH11]. A similar technique could be used in PGS repair to spread the routes it produces.

An alternative approach would be to adapt the base routing algorithms used prior to PGS repair to, for example, attempt alternative dimension order routes which may avoid blockages due to faulty links.

Fault and irregularity aware placement

One of the shortcomings of the simulated annealing based placer developed in chapter 6 is that it does not account for network faults or irregularities when

estimating the cost of placement solutions. Future work may exploit techniques used in congestion-aware circuit placement which could be adapted for application placement [VPC07].

7.2.4 Reducing placement execution time

The simulated annealing based placer presented in chapter 6 produced good quality placements but its execution time limits its use beyond one million vertex placement problems. Future work should explore possibilities for improving the performance and scalability of this technique.

In addition to considering alternative placement algorithms based on other methods, one possible approach is to attempt to reduce the execution time of simulated annealing based placement by shrinking the application graph being placed.

For example, graph clustering [Sch07] may be used to group together strongly connected vertices which would then be placed as a single unit. Unfortunately, clustering can suffer from the same problems as graph partitioning based placement: vertices may be grouped together in ways which, in practice, cannot be packed together into a given portion of a machine. A possible solution to this problem is to use a two-phase placement approach [KLMH11]. In a 'global' placement phase, solutions are permitted which can slightly over-allocate resources but overall achieve good placement quality. In the 'detailed' placement phase which follows, the solution is 'legalised' by making small changes to the global placement to eliminate over-allocation.

An alternative approach suited to SpiNNaker could be to limit the clustering process to clusters which fit on a single SpiNNaker chip. In typical SpiNNaker application graphs, clustering to this level may reduce placement problem sizes by an order of magnitude and, consequently, reduce execution times by the

same ratio. Preliminary experiments suggest that this approach might result in little placement quality loss for large placement problems whilst substantially reducing overall execution time.

7.2.5 Benchmarking

One of the most significant limitations of this study has been the unavailability of large-scale SpiNNaker applications for use as benchmarks. As a consequence, much of the scalability experimentation performed has relied on simple synthetic benchmarks based on projections of future application behaviour.

In the short term, more sophisticated synthetic benchmark generation techniques used by the circuit placement community [NC07] may offer alternative benchmarks for future work. In the longer term, however, it is hoped that the availability of large SpiNNaker systems, and placement and routing algorithms better suited to exploit them, will lead to larger scale applications being developed. Hopefully these applications will lead to more interesting and representative benchmarks for use in future work.

7.3 Closing remarks

One of the primary outcomes of this work is that a number of the practical challenges faced in scaling up the SpiNNaker architecture have been addressed, leading to the construction of large-scale SpiNNaker machines. The development of an effective placement algorithm for SpiNNaker applications has been shown to enable some neural simulations to exploit SpiNNaker's architecture for the first time. The availability of larger SpiNNaker machines paves the way for future large-scale neural modelling work built on much larger models such as Spaun, 'the world's largest functional brain model' [ESC+12].

Beyond the SpiNNaker project, the hexagonal torus topology has also been validated as a scalable and practical candidate for future network architectures. As supercomputers become ever larger, the physical scalability afforded by the 2D nature of the hexagonal torus topology may make it a compelling option. In addition, the finding that circuit placement techniques can be adapted to support placement of SpiNNaker software indicates that these algorithms may also be applicable to other applications. Indeed, if this is the case, circuit placement may offer a long-term source of placement algorithms able to handle the demands of future applications.

Although this work stops short of demonstrating truly large-scale neuroscientific simulations running at the scale of newly completed SpiNNaker machines (largely because such simulations do not yet exist) a number of smaller-scale neural simulations have been made possible for the first time. The algorithms and techniques devised in this work have subsequently been incorporated into various software libraries and tools now being used by researchers building SpiNNaker applications, vindicating the efforts of this thesis (see appendix C). A successor to the SpiNNaker architecture is also in the early stages of design and is building on experience of the existing architecture. The current intention is to retain the hexagonal torus topology used by SpiNNaker, a decision supported by the findings of this thesis.

With SpiNNaker's hardware architecture now operating at scales close to its architectural limits, it is hoped that the contributions of this work will enable researchers to develop larger and more detailed neural models for this unique architecture.

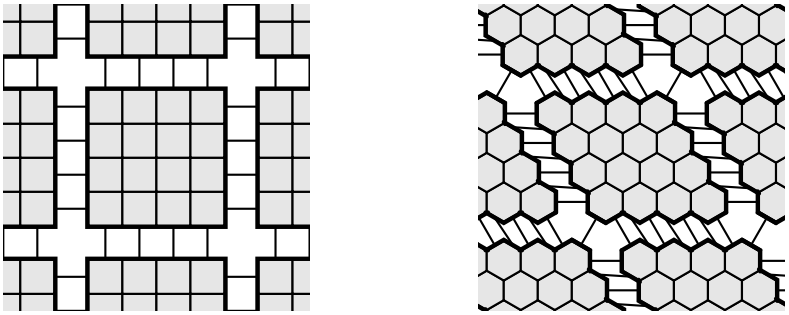
Appendix A

Partitioning hexagonal toruses

The nodes in supercomputer networks are usually relatively small, for example in SpiNNaker each node is a single chip. To allow several nodes to share resources such as power supplies and to simplify construction, tens of nodes are typically packed together into a single unit such as a circuit board or server blade [Gil14, AIHS12]. In commercial supercomputers built on non-hexagonal torus topologies, units usually represent a hypercube partition of the overall topology as illustrated in figure A.1a [CEH+11, AIHS12].

The analogue of this scheme in a hexagonal torus topology is a parallelogram as illustrated in figure A.1b. Each partition connects to six neighbouring partitions and, unlike hypercube partitions, the number of connections to each is imbalanced. Specifically the partitions above-right and below-left are connected by only one link each while others are connected by many. As a consequence, multiple interconnect technologies may be needed if connections between partitions are concentrated into single cables as in SpiNNaker (see chapter 2). This adds both to design complexity and system cost.

In this appendix, I describe how, and why, the 'wrapped triple' partitioning scheme devised by Davidson for SpiNNaker works [Dav13].



(a) 2D hypercube partitioning (b) Parallelogram partitioning

Figure A.1: Partitioning schemes for 2D and hexagonal torus topologies. (Duplicate of figures 3.3a and 3.3b.)

A.1 Tiling

For a particular configuration of nodes to form a valid partition, it must be possible to use this configuration to 'tile' a hexagonal torus. In this thesis, a 'pointy-topped' hexagon is used to represent a chip in a hexagonal torus which is drawn (by convention) in a parallelogram-shaped arrangement.

Any partition which shares the 'translational symmetry' of a pointy-topped hexagon must also tile a hexagonal torus. For example, in figure A.2 we can see that 2×2 parallelograms can tile a 9×9 hexagonal torus topology. In figure A.3, I demonstrate that this parallelogram partition shares its translational symmetry with a pointy-topped hexagon. In figure A.3a, the 2×2 partition is shown with each node shaded differently and, in figure A.3b, a pointy-topped hexagon is superimposed. By tiling several copies of the partition (figure A.3c), we can see that the repeating pattern of the parallelograms matches that of the pointy-topped hexagons. I refer to this property as the parallelogram sharing the translational symmetry of a pointy-topped hexagon.

Notice that the parallelogram partition can be redrawn so that parts protruding from the overlaid hexagon wrap around to fill the gaps on opposite sides producing a pointy-topped hexagon shaped tile as shown in figure A.3d.

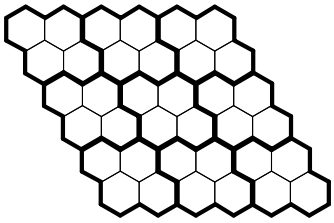
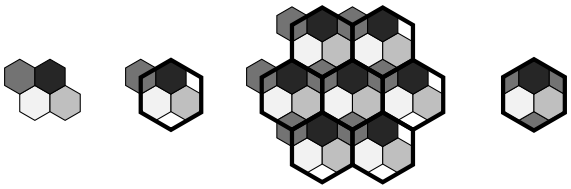


Figure A.2: Tiling a hexagonal torus with parallelograms.



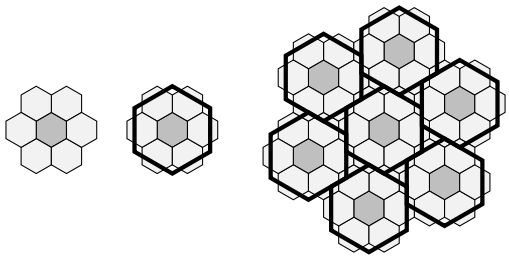
(a) (b) (c) (d)

Figure A.3: Visual proof that a parallelogram shares translational symmetry with a pointy-topped hexagon.

A.2 How *not* to tile a hexagonal torus

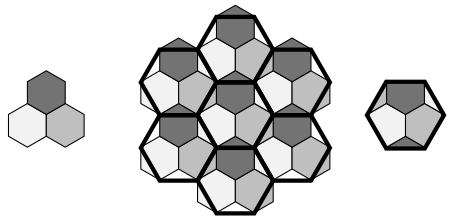
An 'obvious' partitioning scheme for a hexagonal topology, which evenly distributes links between six sides, is a hexagonal partition. Such a partition might naïvely be formed by wrapping 'layers' of hexagons around a central hexagon as illustrated in figure A.4a.

While this type of partition exposes six equally-sized edges (satisfying the requirement that connections between partitions should have a balanced num-



(a) (b) (c)

Figure A.4: Visual proof that a wrapped hexagon does not share translational symmetry with a pointy-topped hexagon.



(a) (b) (c)

Figure A.5: Visual proof that a triple of pointy-topped hexagons shares the same translational symmetry as a flat-topped hexagon.

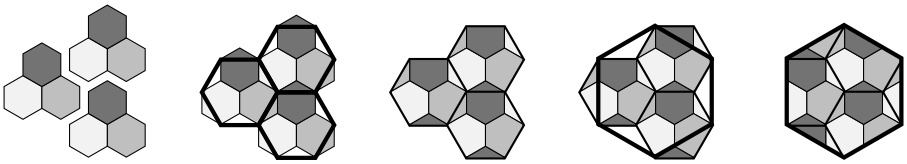
ber of connections) this partition does not share translational symmetry with a pointy-topped hexagon. Figure A.4b shows a best fitting pointy-topped hexagon superimposed on the partition. In figure A.4c, we can see that when tiled, the partition leaves gaps between the superimposed pointy-topped hexagons indicating it does not share translational symmetry.

Consequently, it is not possible to construct a hexagonal torus from partitions of this shape. As an aside, it *is* possible to construct a H-torus topology [ZYW08], a related, but different, topology mentioned briefly in §2.5.

A.3 Triads of triples

A 'triple' is a partition made up of three nodes arranged as in figure A.5a. This partition's edges may be broken into six groups with an equal number of connections, meeting the requirements set out at the beginning of this appendix. A triple, however, does not share translational symmetry with a pointy-topped hexagon but *does* share it with a 'flat-topped' hexagon (the 30°-rotated cousin of the pointy-topped hexagon) as demonstrated in figures A.5b and A.5c.

Because a triple made up of pointy-topped hexagons shares translational symmetry with a flat-topped hexagon, a triple made up of flat-topped hexagons must also share translational symmetry with a pointy-topped hexagon (turn



(a) (b) (c) (d) (e)

Figure A.6: Demonstration that a triad shares translational symmetry with a pointy-topped hexagon.

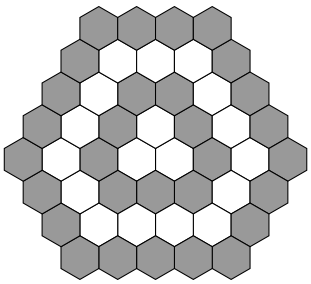
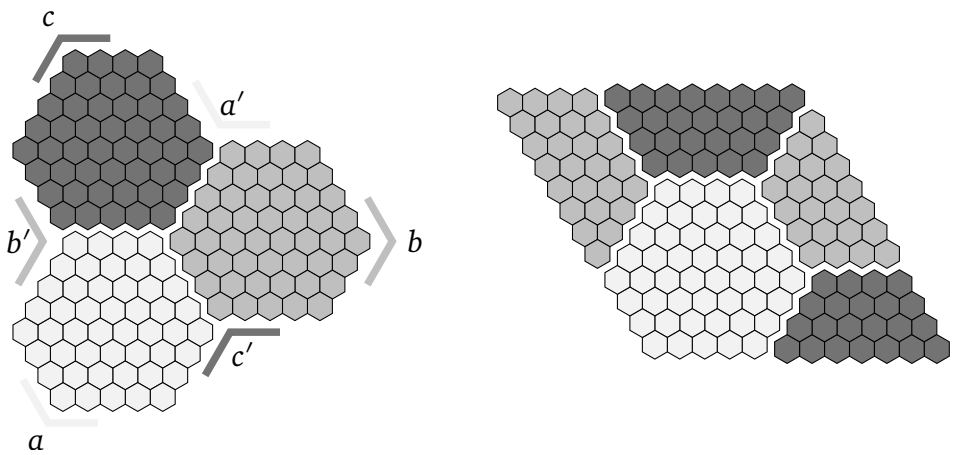


Figure A.7: A wrapped triple with four layers. Layers are shaded alternately.

figure A.5 by 30° to verify this). Therefore, it follows that three triples arranged as in figure A.6 – a 'triad' – share translational symmetry with pointy-topped hexagons. Consequently, triads of triples may be used to tile a hexagonal torus topology.

A.4 Wrapped triples

A triple forms a partition which can be used to tile a hexagonal torus when tiled using triads of triples. To increase the number of nodes in the partition, layers of hexagons may be wrapped around a triple to form a 'wrapped triple' as in figure A.7. Wrapped triples have the same balanced communications as triples and triads of wrapped triples also share translational symmetry with a pointy-topped hexagon. Therefore, triads of wrapped triples may be used to tile a hexagonal torus topology.



(a) A triad of boards

(b) . . . as a parallelogram

Figure A.8: Logical connectivity of chips (hexagons) in a triad of SpiNNaker boards. (Duplicate of figure 2.11.)

In SpiNNaker, each board partitions the network into a four-layer wrapped triple (figure A.7) containing forty eight chips (nodes) each, convenient for implementation on a standard-sized 220.00 × 233.35 mm Eurocard circuit board [IEE98].

A triad of boards may be interconnected to form a 12 × 12 hexagonal torus as illustrated in figure A.8. Six boards may be interconnected to form a 12 × 24 or 24 × 12 topology. Nine can form a 12 × 36 or 36 × 12 topology. At twelve boards, a 'square' 24 × 24 topology may be formed, corresponding to half a frame in a large SpiNNaker installation. The architectural limit of the SpiNNaker architecture combines 1200 boards to produce a 240 × 240 hexagonal torus topology.

Appendix B

Minimising hexagonal mesh coordinates

As in non-hexagonal mesh and torus topologies, coordinates in a hexagonal torus topology are given in terms of the offset of a point along each axis of the topology from an arbitrary origin. Hexagonal topologies, unusually, are considered as having three, non-orthogonal axes. In a conventional 2D topology, for example, moving along the X axis does not alter your position on the Y axis because the X and Y axes are orthogonal. In a hexagonal coordinate system, moving along any one of the three axes results in some movement in the other two. A consequence of this is that multiple coordinates may exist for the same position.

Consider the vector $(1, 1, 1)$ which takes one hop along each axis. Figure B.1 illustrates how this vector takes us back to our starting point after three hops. Because, like $(0, 0, 0)$, this vector results in no movement, it may be added or subtracted to any existing vector or coordinate without changing the meaning of the vector. Doing so, however, clearly has some impact on the magnitude of the vector. For example, the coordinate vector $(2, -3, -1)$ has a

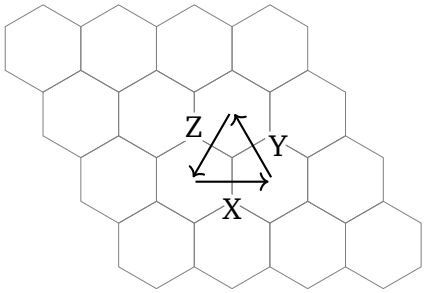


Figure B.1: The vector $(1, 1, 1)$ in a hexagonal mesh topology results in no movement.

magnitude of 6. Adding $(1, 1, 1)$ results in a new, equivalent vector, $(3, -2, 0)$, with a reduced magnitude of 5. Adding $(1, 1, 1)$ once more produces the vector $(4, -1, 1)$ of magnitude 6 again.

In a hexagonal mesh, every point has a unique coordinate vector whose magnitude is minimal. To demonstrate this by cases, consider:

- If a vector has three positive elements, subtracting $(1, 1, 1)$ *reduces* the magnitude of the vector by three overall.
- If a vector has two positive elements, subtracting $(1, 1, 1)$ decreases the magnitude of those two elements and increases the magnitude of the remaining element resulting in a net *reduction* in magnitude of one.
- If a vector has only one positive, non-zero element, subtracting $(1, 1, 1)$ decreases the magnitude of that element and increases the magnitude of the remaining two resulting in a net *increase* in magnitude of one.
- Similar arguments may be made for vectors with negative elements and the *addition* of $(1, 1, 1)$.

These cases indicate that vectors and coordinates containing at least one zero element with the remaining elements having opposite signs are minimal

since adding or subtracting $(1,1,1)$ will always increase the magnitude of the vector.

To minimise a coordinate vector in a hexagonal mesh topology, the following function may be used, producing a unique coordinate for any given point:

$$\text{minimiseVector}(x,y,z) = (x,y,z) - \text{median}(x,y,z) \cdot (1,1,1)$$

As an aside, minimising a hexagonal coordinate vector is not the only way to determine a unique coordinate for a given point. Given a vector of the form (x,y,z), subtracting (z,z,z) will result in a vector of the form $(x',y',0)$. Because this form mimics the appearance of a standard 2D coordinate system, while uniquely identifying points, it is widely used as a convenient and intuitive addressing scheme by SpiNNaker's system software.

Appendix C

Software

I have implemented several of the techniques described in this thesis in various software systems which are now in use in the wider SpiNNaker ecosystem. These implementations may prove instructive to future researchers hoping to take advantage of or extend the techniques described by this work.

SpiNNer: A collection of tools for generating cabling plans and guiding cable installation and maintenance of SpiNNaker machines using the techniques described in chapter 3.

Source	`https://github.com/SpiNNakerManchester/SpiNNer`
Manual	`http://spinner.readthedocs.org/`

Rig: A library for SpiNNaker application writers which, among other features, implements the shortest-path vector geometry functions introduced in chapter 4, the NER/PGS repair routing algorithm described in chapter 5 and the simulated annealing based placement algorithm from chapter 6. Written in collaboration with Andrew Mundy and Jamie Knight.

Source	`https://github.com/project-rig/rig`
Manual	`http://rig.readthedocs.org/`

Rig C SA: A C implementation of the simulated annealing based placement algorithm from chapter 6. An optional drop-in replacement for Python implementation of the algorithm in the Rig library.

Source `https://github.com/project-rig/rig_c_sa`

Network Tester: A library for quickly building and running experiments on SpiNNaker's network and used to perform some of the experiments in chapters 5 and 6.

Source `https://github.com/SpiNNakerManchester/network_tester`
Manual `http://network-tester.readthedocs.org/`

Spalloc: A centralised system which partitions large SpiNNaker machines into smaller ones on demand. This system was used to perform many experiments in parallel. This system is also widely used by researchers to share access to large SpiNNaker machines.

Source `https://github.com/SpiNNakerManchester/spalloc_server`
`https://github.com/SpiNNakerManchester/spalloc`
Manual `http://spalloc-server.readthedocs.org/`
`http://spalloc.readthedocs.org/`

In addition to these 'stand-alone' software packages, the source code and raw result data for the experiments in this thesis are available to download from GitHub:

`https://github.com/mossblaser/phd_thesis`
`https://github.com/mossblaser/phd_thesis_experiments`

Bibliography

[AAA+02] N. R. Adiga, G. Almasi, G. S. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, et al. An overview of the BlueGene/L supercomputer. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 60–82. IEEE, 2002.

[AC05] Abhijit Athavale and Carl Christensen. *High-speed serial I/O made simple*. Xilinx Inc, 2005.

[AFKR12] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray XC series network. *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[AIHS12] Yuuichirou Ajima, Tomohiro Inoue, Shinya Hiramoto, and Toshiyuki Shimizu. Tofu: Interconnect for the K computer. *Fujitsu Sci. Tech. J*, 48(3):280–285, 2012.

[Ami15] Amit Patel. Hexagonal grids. `http://www.redblobgames.com/grids/hexagons/`, 2015. Accessed 27 January 2015.

[Ass06] Telecommunication Industry Association. Ansi/tia-942: Data center standards, 2006.

[Aub15] Sean R. Aubin. A biologically plausible model of the Wisconsin card sorting test. Technical report, Nengo Summer School 2015, May 2015.

[BBH+13] Trevor Bekolay, James Bergstra, Eric Hunsberger, Travis DeWolf, Terrence C Stewart, Daniel Rasmussen, Xuan Choo, Aaron Russell Voelker, and Chris Eliasmith. Nengo: a Python tool for building large-scale functional brain models. *Frontiers in neuroinformatics*, 7, 2013.

[BC95] Rajendra V Boppana and Suresh Chalasani. Fault-tolerant wormhole routing algorithms for mesh networks. *Computers, IEEE Transactions on*, 44(7):848–864, 1995.

[BGM+14] Ben Varkey Benjamin, Peiran Gao, Emmett McQuinn, Swadesh Choudhary, Anand R Chandrasekaran, J Bussat, R Alvarez-Icaza, JV Arthur, PA Merolla, and K Boahen. Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations. *Proceedings of the IEEE*, 102(5):699–716, 2014.

[BMBL+13] Iván Bermejo-Moreno, Julien Bodart, Johan Larsson, Blaise M Barney, Joseph W Nichols, and Steve Jones. Solving the compressible Navier-Stokes equations on up to 1.97 million cores and 4.1 trillion grid points. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, page 62. ACM, 2013.

[BR97] Vaughn Betz and Jonathan Rose. VPR: A new packing, placement and routing tool for FPGA research. In *Field-Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[Bre77] Melvin A Breuer. A class of min-cut placement algorithms. In *Proceedings of the 14th Design Automation Conference*, pages 284–290. IEEE Press, 1977.

[CBZ10] Alexander Choong, Rami Beidas, and Jianwen Zhu. Parallelizing simulated annealing-based placement using GPGPU. In *Field Programmable Logic and Applications (FPL), 2010 International Conference on*, pages 31–34. IEEE, 2010.

[CCE+12] Andrew R Curtis, Tommy Carpenter, Mustafa Elsheikh, Alejandro López-Ortiz, and Srinivasan Keshav. Rewire: An optimization-based framework for unstructured data center network design. In *INFOCOM, 2012 Proceedings IEEE*, pages 1116–1124. IEEE, 2012.

[CCWS11] George Chiu, Paul Coteus, Robert Wisniewski, and Jim Sexton. Blue Gene/Q Overview and Update. `https://www.alcf.anl.gov/files/IBM_BGQ_Architecture_0.pdf`, November 2011. Accessed 12 May 2016.

[CEH+11] Dong Chen, Noel A Eisley, Philip Heidelberger, Robert M Senger, Yutaka Sugawara, Sameer Kumar, Valentina Salapura, David L Satterfield, Burkhard Steinmacher-Burow, and Jeffrey J Parker. The IBM Blue Gene/Q interconnection network and message unit. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–10. IEEE, 2011.

[CH79] FRK Chung and FK Hwang. The largest minimal rectilinear steiner trees for a set of n points enclosed in a rectangle with given perimeter. *Networks*, 9(1):19–36, 1979.

[Cis07] Cisco. Cisco data center infrastructure 2.5 design guide, 2007.

[CM69] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM, 1969.

[CMV+10] José M Cámara, Miquel Moretó, Enrique Vallejo, Ramón Beivide, José Miguel-Alonso, Carmen Martínez, and Javier Navaridas. Twisted torus topologies for enhanced interconnection networks. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1765–1778, 2010.

[Com01] SFF Committee. INF-8074i specification for SFP (Small Formfactor Pluggable) transceiver. *Rev*, 1:1–38, 2001.

[CSC13] CSCS (Swiss National Super Computing Center). Installation of an IBM BlueGene/Q supercomputer at CSCS for EPFL BlueBrain project. `https://www.youtube.com/watch?v=XHxJijhyQE4`, 2013. Accessed 29 Febuary 2016.

[Cse15] Márton Csernai. *Efficient cabling in data center networks*. PhD thesis, Budapesti Műszaki és Gazdaságtudományi Egyetem, 2015.

[Cse16] Cseed. Arachne-pnr. `https://github.com/cseed/arachne-pnr`, 2016. Accessed 06 June 2016.

[Dal87] William J Dally. Concurrent computer architecture. Technical report, DTIC Document, 1987.

[Dav13] Simon Davidson. Summary of building a toroid using hexagons, v1.2. An internal document describing how chips are distributed between boards and how boards can be linked to form a toroid., 2013.

[DBE+08] Andrew P Davison, Daniel Brüderle, Jochen Eppler, Jens Kremkow, Eilif Muller, Dejan Pecevski, Laurent Perrinet, and Pierre Yger. PyNN: a common interface for neuronal network simulators. *Frontiers in neuroinformatics*, 2, 2008.

[DNGF12] Sergio Davies, Javier Navaridas, Francesco Galluppi, and Steve B Furber. Population-based routing in the SpiNNaker neuromorphic architecture. In *Neural Networks (IJCNN), The 2012 International Joint Conference on*, pages 1–8. IEEE, 2012.

[Don16] Jack Dongarra. Report on the Sunway TaihuLight system, June 2016.

[DT01] William J Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. In *Design Automation Conference, 2001. Proceedings*, pages 684–689. IEEE, 2001.

[DT04] William J Dally and Brian Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2004.

[EA04] Chris Eliasmith and Charles H Anderson. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. MIT Press, 2004.

[Eli13] Chris Eliasmith. *How to build a brain: A neural architecture for biological cognition*. Oxford University Press, 2013.

[ESC+12] Chris Eliasmith, Terrence C Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Charlie Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *Science*, 338(6111):1202–1205, 2012.

[ET14] Chris Eliasmith and Oliver Trujillo. The use and abuse of large-scale brain models. *Current Opinion in Neurobiology*, 25:1–6, 2014.

[FDC+09] David Fick, Andrew DeOrio, Gregory Chen, Valeria Bertacco, Dennis Sylvester, and David Blaauw. A highly resilient routing algorithm for fault-tolerant NoCs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 21–26. European Design and Automation Association, 2009.

[FLP+13] Steve B Furber, David R Lester, Luis A Plana, Jim D Garside, Eustace Painkras, Steve Temple, and Andrew D Brown. Overview of the SpiNNaker system architecture. *IEEE Transactions on Computers*, 62(12):2454–2467, 2013.

[FT07] Steve B Furber and Steve Temple. Neural systems engineering. *Journal of the Royal Society interface*, 4(13):193–206, 2007.

[FTB06] Steve B Furber, Steve Temple, and AD Brown. High-performance computing for systems of spiking neurons. In *AISBâĂŹ06 workshop on GC5: Architecture of Brain and Mind*, volume 2, pages 29–36, 2006.

[GBC+05] Alan Gara, Matthias A Blumrich, Dong Chen, GL-T Chiu, Paul Coteus, Mark E Giampapa, Ruud A Haring, Philip Heidelberger, Dirk Hoenicke, Gerard V Kopcsay, et al. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, 2005.

[GD07] Marc-Oliver Gewaltig and Markus Diesmann. NEST (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[GDF+04] Maria E Gomez, Jose Duato, Jose Flich, Pedro Lopez, Antonio Robles, Nils Agne Nordbotten, Olav Lysne, and Tor Skeie. An efficient fault-tolerant routing methodology for meshes and tori. *Computer Architecture Letters*, 3(1):3–3, 2004.

[GHN+15] Mohsen Ghasempour, Jonathan Heathcote, Javier Navaridas, Luis A Plana, Jim Garside, and Mikel Luján. Analysis of FPGA and software approaches to simulate unconventional computer architectures. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2015.

[Gil14] Megan Gilge. *IBM System Blue Gene solution: Blue Gene/Q application development*. IBM Redbooks, 2014.

[GRDF10] Francesco Galluppi, Alexander Rast, Sergio Davies, and Steve B Furber. A general-purpose model translation system for a universal neural chip. In *Neural information processing. Theory and algorithms*, pages 58–65. Springer, 2010.

[GWT+08] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. Dcell: a scalable and fault-tolerant network structure for data centers. *ACM SIGCOMM Computer Communication Review*, 38(4):75–86, 2008.

[HD11] Rolf Hoffmann and Dominique Désérable. Efficient minimal routing in the triangular grid with six channels. In *Parallel Computing Technologies*, pages 152–165. Springer, 2011.

[HD15] Rolf Hoffmann and Dominique Désérable. Routing by cellular automata agents in the triangular lattice. In Georgios Ch Sir-

akoulis and Andrew Adamatzky, editors, *Robots and Lattice Automata*, chapter 6, pages 117–147. Springer, Switzerland, 2015.

[Hea16a] Jonathan Heathcote. Building and wiring up the half-million core SpiNNaker machine. `https://www.youtube.com/watch?v=z1_gE_ugEgE`, 2016. Accessed 27 June 2016.

[Hea16b] Jonathan Heathcote. Spalloc server. `https://github.com/project-rig/spalloc_server`, 2016. Accessed 16 July 2016.

[Hil91] David Hilbert. Ueber die stetige Abbildung einer Line auf ein Flächenstück. *Mathematische Annalen*, 38(3):459–460, 1891. Though written in the German language (which the author of this thesis does not speak), this short paper centers around an informative illustration which explains the construction and properties of the Hilbert curve.

[HMK15] Jonathan Heathcote, Andrew Mundy, and Jamie Knight. Rig. `https://github.com/project-rig/rig`, 2015. Accessed 16 July 2016.

[HNCB00] Malay Haldar, Anshuman Nayak, Alok Choudhary, and Prith Banerjee. Parallel algorithms for FPGA placement. In *Proceedings of the 10th Great Lakes symposium on VLSI*, pages 86–94. ACM, 2000.

[HS04] Ching-Tien Ho and Larry Stockmeyer. A new approach to fault-tolerant wormhole routing for mesh-connected parallel computers. *Computers, IEEE Transactions on*, 53(4):427–438, 2004.

[HS11] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the*

international conference on Supercomputing, pages 75–84. ACM, 2011.

[IEE98] IEEE-SA Standards Board. *IEEE Standard for Mechanical Core Specifications for Microcomputers Using IEC 60603-2 Connectors*. IEEE Standard 11.01.1-1998 edition, 1998.

[JMT14] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *Parallel and Distributed Systems, IEEE Transactions on*, 25(4):993–1002, 2014.

[KGV+83] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[KLM12] Myung-Chul Kim, Dong-Jin Lee, and Igor L Markov. SimPL An effective placement algorithm. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 31(1):50–60, 2012.

[KLMH11] Andrew B Kahng, Jens Lienig, Igor L Markov, and Jin Hu. *VLSI physical design: from graph partitioning to timing closure*, volume 312. Springer, 2011.

[KTK+16] James C Knight, Philip J Tully, Bernhard A Kaplan, Anders Lansner, and Steve B Furber. Large-scale simulations of plastic neural networks on neuromorphic hardware. *Frontiers in neuroanatomy*, 10, 2016.

[LBP08] Adrian Ludwin, Vaughn Betz, and Ketan Padalia. High-quality, deterministic parallel placement for FPGAs on commodity hardware. In *Proceedings of the 16th international ACM/SIGDA symposium on Field programmable gate arrays*, pages 14–23. ACM, 2008.

[LGW+14] Jason Luu, Jeff Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Norrudin Ahmed, Kenneth B. Kent, Jason Anderson, Jonathan Rose, and Vaughn Betz. VTR 7.0: Next Generation Architecture and CAD System for FPGAs. *ACM Transactions on Reconfigurable Technology and Systems*, 7(2):6:1–6:30, June 2014.

[LMMS07] Gary Lakner, James Milano, and Gary Mullen-Schultz. *IBM System Blue Gene Solution: Hardware Installation and Serviceability*. Vervante, 2007.

[LYYG12] Yong Liao, Jiangtao Yin, Dong Yin, and Lixin Gao. Dpillar: Dual-port server interconnection network for large scale data centers. *Computer Networks*, 56(8):2132–2147, 2012.

[Maz97] Dennis Mazaris. The reality of patch-cord management. *Cabling Installation & Maintainance*, Febuary 1997.

[Meu16] Hans Meuer. Top500 List - June 2016. `http://www.top500.org/list/2016/06/`, June 2016. Accessed 5 July 2016.

[MFD+06] Andres Mejia, Jose Flich, Jose Duato, Sven-Arne Reinemo, and Tor Skeie. Segment-based routing: an efficient fault-tolerant routing algorithm for meshes and tori. In *Parallel and Distributed*

Processing Symposium, 2006. IPDPS 2006. 20th International, pages 10–pp. IEEE, 2006.

[MFM+12] Simon W Moore, Paul J Fox, Steven JT Marsh, A Theodore Markettos, and Alan Mujumdar. Bluehive-a field-programable custom computing machine for extreme-scale real-time neural network simulation. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 133–140. IEEE, 2012.

[MHG16] Andrew Mundy, Jonathan Heathcote, and Jim Garside. On-chip order-exploiting routing table minimization for a multicast supercomputer network. In *2016 IEEE 17th International Conference on High Performance Switching and Routing (HPSR)*, 2016.

[MHK15] Igor L Markov, Jin Hu, and Myung-Chul Kim. Progress and challenges in VLSI placement research. *Proceedings of the IEEE*, 103(11):1985–2003, 2015.

[MJFS01] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):124–141, 2001.

[MKSF15] Andrew Mundy, James Knight, Terrence C Stewart, and Steve B Furber. An efficient SpiNNaker implementation of the neural engineering framework. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8. IEEE, 2015.

[ML13] James Milano and Pamela Lembke. *IBM system Blue Gene solution: Blue Gene/Q hardware overview and installation planning*. IBM Redbooks, 2013.

[Moo75] Gordon Moore. Progress in digital integrated electronics. In *Electron Devices Meeting*, volume 21, pages 11–13, 1975.

[MS10] Janardan Misra and Indranil Saha. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(1-3):239–255, 2010.

[MYM11] Jayaram Mudigonda, Praveen Yalagandula, and Jeffrey C Mogul. Taming the flying cable monster: A topology design and optimization framework for data-center networks. In *USENIX Annual Technical Conference*, 2011.

[NC07] Gi-Joon Nam and Jingsheng Jason Cong. *Modern circuit placement: best practices and results*. Springer, 2007.

[NLMA+09] Javier Navaridas, Mikel Luján, Jose Miguel-Alonso, Luis A Plana, and Steve B Furber. Understanding the interconnection network of SpiNNaker. In *Proceedings of the 23rd international conference on Supercomputing*, pages 286–295. ACM, 2009.

[NLP+12] Javier Navaridas, Mikel Luján, Luis A Plana, Jose Miguel-Alonso, and Steve B Furber. Analytical Assessment of the Suitability of Multicast Communications for the SpiNNaker Neuromimetic System. In *High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 1–8. IEEE, 2012.

[NLP+15] Javier Navaridas, Mikel Luján, Luis A Plana, Steve Temple, and Steve B Furber. SpiNNaker: enhanced multicast routing. *Parallel Computing*, 45(0):49 – 66, 2015.

[NPMA09] Javier Navaridas, Jose Antonio Pascual, and Jose Miguel-Alonso. Effects of job and task placement on parallel scientific applications performance. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 55–61. IEEE, 2009.

[Oak16] Oak Ridge National Laboratory. Introducing Titan. `https://www.olcf.ornl.gov/titan/`, 2016. Accessed 6 July 2016.

[Pau16] Paul Nutter. COMP12111 Fundamentals of computer engineering. `http://studentnet.cs.manchester.ac.uk/ugt/2016/COMP12111/syllabus/`, 2016. Accessed 16 June 2016.

[PD14] Tobias C Potjans and Markus Diesmann. The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model. *Cerebral Cortex*, 24(3):785–806, 2014.

[PG07] Valentin Puente and Jose Angel Gregorio. Immucube: Scalable fault-tolerant routing for k-ary n-cube networks. *Parallel and Distributed Systems, IEEE Transactions on*, 18(6):776–788, 2007.

[PR96] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[PRI+10] Lucian Popa, Sylvia Ratnasamy, Gianluca Iannaccone, Arvind Krishnamurthy, and Ion Stoica. A cost comparison of datacenter

network architectures. In *Proceedings of the 6th International Conference*, page 16. ACM, 2010.

[PSJ+14] James C Phillips, Yanhua Sun, Nikhil Jain, Eric J Bohm, and Laxmikant V Kalé. Mapping to irregular torus topologies and other techniques for petascale biomolecular simulation. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 81–91. IEEE Press, 2014.

[RMF+09] Samuel Rodrigo, Simone Medardoni, Jose Flich, Davide Bertozzi, and Jose Duato. Efficient implementation of distributed routing algorithms for NoCs. *Computers & Digital Techniques, IET*, 3(5):460–475, 2009.

[RPA+05] Jarrod A Roy, David A Papa, Saurabh N Adya, Hayward H Chan, Aaron N Ng, James F Lu, and Igor L Markov. Capo: robust and scalable open-source min-cut floorplacer. In *Proceedings of the 2005 international symposium on Physical design*, pages 224–226. ACM, 2005.

[SBG+10] Johannes Schemmel, D Bruderle, A Grubl, Matthias Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 1947–1950. IEEE, 2010.

[Sch07] Satu Elisa Schaeffer. Graph clustering. *Computer Science Review*, 1(1):27–64, 2007.

[Ser08] Serial ATA International Organisation. Serial ATA Revision 3.0. https://www.sata-io.org/, 2008. Accessed 26 June 2014.

[SSJ08] Peter Spindler, Ulf Schlichtmann, and Frank M Johannes. Kraftwerk2 – a fast force-directed quadratic placement approach using an accurate net model. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(8):1398–1411, 2008.

[SSV85] Carl Sechen and Alberto Sangiovanni-Vincentelli. The TimberWolf placement and routing package. *IEEE Journal of Solid-State Circuits*, 20(2):510–522, 1985.

[ST97] Horst D Simon and Shang-Hua Teng. How good is recursive bisection? *SIAM Journal on Scientific Computing*, 18(5):1436–1445, 1997.

[Ste85] Craig S Steele. Placement of communicating processes on multiprocessor networks. Master's thesis, California Institute of Technology, 1985.

[THO13] THOCP. Cray 1. http://www.thocp.net/hardware/cray_1.htm, March 2013. Accessed 29 September 2016.

[Trä02] Jesper Larsson Träff. Implementing the MPI process topology mechanism. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 28–28. IEEE, 2002.

[VG11] Dmitri Vainbrand and Ran Ginosar. Scalable network-on-chip architecture for configurable neural networks. *Microprocessors and Microsystems*, 35(2):152–166, 2011.

[VPC07] Natarajan Viswanathan, Min Pan, and Chris Chu. Fastplace 3.0: A fast multilevel quadratic placement algorithm with placement congestion control. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, pages 135–140. IEEE Computer Society, 2007.

[XH91] Jian Xu and Kai Hwang. Mapping rule-based systems onto multicomputers using simulated annealing. *Journal of Parallel and Distributed Computing*, 13(4):442–455, 1991.

[XP04] Wenjun Xiao and Behrooz Parhami. Hexagonal and pruned torus networks as Cayley graphs. In *In Proc. International Conf. Communications in Computing, Las Vegas, NV*, pages 107–112, 2004.

[YJG03] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[Zum99] G Zumbusch. A parallel adaptive multigrid method. In *Proceedings of the 15th GAMM-Seminar Kiel on Numerical Techniques for Composite Materials*, 1999.

[ZYW08] You-Jian Zhao, Zu-Hui Yue, and Jian-Ping Wu. Research on next-generation scalable routers implemented with H-torus topology. *Journal of Computer Science and Technology*, 23(4):684–693, 2008.