UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

Non-Neural Computing on the SpiNNaker Neuromorphic Computer

by

Kier J. Dugan

Thesis for the degree of Doctor of Philosophy

July 2016

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING
Electronics and Computer Science

Doctor of Philosophy

NON-NEURAL COMPUTING ON THE SPINNAKER NEUROMORPHIC COMPUTER

by Kier J. Dugan

Moore's law scaling has slowed dramatically since the turn of the millennium, causing new generations of computer hardware to include more processor cores to offer more performance. Desktop computers, server machines, and even mobile phones are all multi-core devices now, and this trend has shown no signs of slowing soon. Eventually, computers will contain so many cores that they will be an *abundant resource*. Using this many processors requires new ways of thinking about software. Biology leads computer architecture here: mammalian brains contain *billions* of neurons embedded in a dense fabric of synapses—the human brain contains about 10^{11} neurons and 10^{15} synapses. Each neuron is essentially a small processing element in its own right.

Neuromorphic hardware draws inspiration from this and is typically used to support neural network simulations. SpiNNaker is one such platform, designed to support simulations containing up to 10^9 neurons and 10^{12} synapses (about 1% of a human brain) in *biological real-time*. This is achieved by embedding a million ARM processors in a bespoke interconnection fabric which is non-deterministic, modelled after spiking neural networks, and predicated on the inherent fault-tolerance present in biological systems.

This thesis uses SpiNNaker as a test-bed for massively-parallel non-neural applications, showing how very fine-grain parallel software can be structured to solve real-world problems. First, we address the inherent non-determinism of the underlying platform, by designing a set of algorithms that discover the topology of an arbitrary SpiNNaker-like machine so that fine-grain parallel software can be mapped onto it. These algorithms are verified against various fault conditions, and remove a shortcoming present in the existing system. Secondly, we demonstrate a fine-grain parallel application, by solving two-dimensional heat-diffusion where *each point of the problem grid* is essentially a self-contained program. This software architecture is subject to various fault conditions to demonstrate the resilience of the approach.

Contents

List of Figures

List of Tables

List of Listings

Declaration of Authorship

I, Kier J. Dugan , declare that the thesis entitled *Non-Neural Computing on the SpiNNaker Neuromorphic Computer* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: Brown, Mills, Reeve, *et al.* (2013) [1], Dugan, Brown, Reeve, *et al.* (2013) [2], Dugan, Reeve, and Brown (2013) [3], Dugan, Reeve, Brown, *et al.* (2014) [4], Brown, Mills, Dugan, *et al.* (2014) [5], Brown, Furber, Reeve, *et al.* (2014) [6], and Brown, Mills, Dugan, *et al.* (2015) [7].

Signed:.....

Date:.....

Acknowledgements

This thesis is the result of many years of hard work, strange emotions, good times (when things worked), bad times (when they did not), and varied avenues of enquiry that I could not have predicted I would wander down.

My foremost thanks are extended to my supervisor, Dr. Jeff Reeve, who first convinced me to embark on the Ph.D. journey after a particularly successful final-year MEng group project, and has since tolerated my 'impeccable' time-planning. Although he is not responsible for the supervision of my work, I would also like to thank to Prof. Andrew Brown for his advice, input, and ability to always identify the elephant in the room. Dr. Rob Mills and I collaborated on several papers on work included in this thesis, and his advice and impromptu mini-lectures on complexity theory helped me get to grips with the massive scale of a machine as large as SpiNNaker. Prof. John Chad helped me appreciate just how similar neural systems and digital systems are, compelling me to ask, perhaps, a few too many questions.

Being a collaborative project, SpiNNaker presented many opportunities to discuss ideas with members of the teams based at the University of Manchester and at the University of Cambridge. I would like to thank Prof. Steve Furber, Dr. Steve Temple, and Dr. Luis Plana from the University of Manchester for their help with and advice on using the SpNNaker platform. I must also thank Prof. Simon Moore, Dr. Theo Markettos, Dr. Paul Fox, and Steve Marsh from the University of Cambridge, whose conversations led down interesting roads, improving my understanding of wherever it was we ended up.

Finally, I would like to thank my parents for putting up with me for the past *mumble* years, and my friends for also putting up with me. Especially whilst I have been working on my Ph.D.; they were amongst the unfortunate few who asked me how I was feeling.

Acronyms and Abbreviations

AER	Address-Event Representation
AHB	Advanced High-performance Bus
API	Application Programming Interface
CAM	Content-Addressable Memory
CMP	Chip Multi-Processor
CPU	Central Processing Unit
DDR	Double Data-Rate
DI	Delay Insensitive
DLT	Device Look-up Table
EOP	End-of-Packet
FPGA	Field Programmable Gate Array
FPU	Floating-Point Unit
FR	Fixed Route
GPGPU	General-Purpose GPU
GPU	Graphical Processing Unit
HDL	Hardware Description Language
IP	Internet Protocol
IPC	Inter-Process Communication
IRQ	Interrupt Request
ITRS	International Technology Roadmap for Semiconductors
LEMS	Low-Entropy Model Specification
LIF	Leaky Integrate-and-Fire
MC	Multicast
MCT	Multicast Table
MIMD	Multiple-Instruction Multiple-Data
MISD	Multiple-Instruction Single-Data
MPI	Message-Passing Interface
MPMD	Multiple-Program Multiple-Data
NEST	Neural Simulation Tool
NN	Nearest Neighbour
NoC	Network-on-Chip
NRZ	Non-Return-to-Zero

NUMA	Non-Uniform Memory Architecture
OOP	Object-Oriented Programming
P2P	Point-to-Point
POR	Power-On Reset
POST	Power-On Self-Test
RAID	Redundant Array of Independent Disks
RAM	Random-Access Memory
ROM	Read-Only Memory
RTL	Register Transfer Language
RTOS	Real-Time Operating System
RTZ	Return-To-Zero
SARK	SpiNNaker Application Run-time Kernel
SCAMP	SpiNNaker Command And Monitoring Program
SCP	SpiNNaker Command Protocol
SDP	SpiNNaker Datagram Protocol
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single-Instruction Multiple-Data
SISD	Single-Instruction Single-Data
SpiNNaker	Spiking Neural-Network Architecture
SPMD	Single-Program Multiple-Data
SRAM	Static Random Access Memory
STDP	Spike Timing-Dependent Plasticity
TCDM / DTCM	Tightly-Coupled Data Memory
TCIM / ITCM	Tightly-Coupled Instruction Memory
TCM	Tightly-Coupled Memory
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UMA	Uniform Memory Architecture
VLIW	Very Long Instruction Word
WFI	Wait For Interrupt
XML	eXtensible Mark-up Language

Chapter 1

Introduction

For the past 50 years, Gordon Moore's 1965 prediction [8] has driven a rapid advance in electronic integrated circuit fabrication technology. Computers have seen a similarly rapid increase in capability because smaller transistors directly lead to an increase in clock speed. With greater clock speed and the ability to fit larger and larger numbers of transistors into a single die, processors were not only able to perform more operations per second, but also able to perform more complex operations.

Despite progress initially slowing ten years after the original prediction [9], planar transistor scaling remains in effect and has led to truly amazing feats of integration such as the Xilinx Virtex7 Ultrascale containing over 20bn transistors [10] and the Oracle SPARC M7 containing over 10bn transistors [11]. Continuing the trend further by scaling transistor gate lengths below 20nm will probably require advances in non-planar fabrication technology to compensate for worsening device parasitics and increased leakage currents [12].

The direct correlation between smaller transistors and increased clock speed was a boon to the software industry and led to great advancements. However, as Figure 1.1 clearly shows, more recent transistor scaling has not brought with it a consequential increase in single-threaded performance. Chip manufacturers have begun to include multiple processor cores in a single package to compensate for the levelling of clock frequency. Observe from Figure 1.1 that frequency scaling slowed around the turn of the millennium, and that the number of cores began increasing around five years afterwards. This shift in system architecture requires new ways of thinking about software which focus on *concurrent execution* rather than conventional single-threaded programs [14].

Whilst modern desktop processors may contain 4–8 cores, some special-purpose architectures have gone further:

- Tilera's TILE64 [15] includes 64 32bit VLIW processors embedded in an 8×8 mesh network that supports a range of static and dynamic routing functions. Each

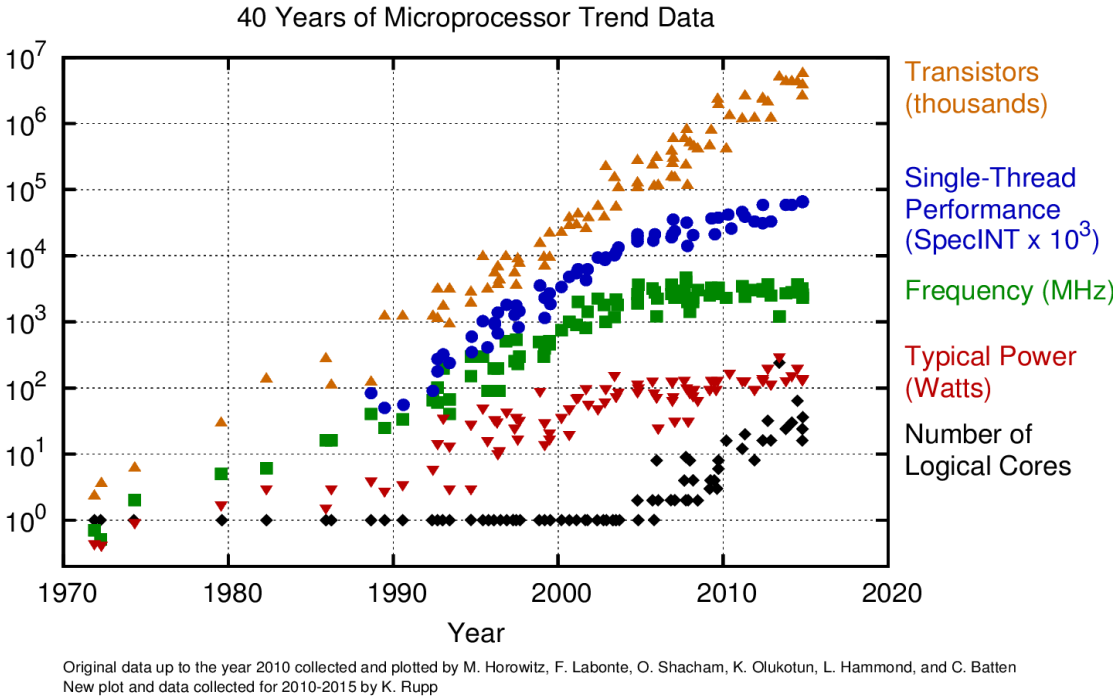


Figure 1.1: 40 years of microprocessor trend data (collated and plotted by Rupp [13]) showing the steady continuation of transistor scaling, but a levelling of clock frequency and single-threaded performance.

processor is resident on a 'tile' which connects to this mesh with a 5-port crossbar-switch-based wormhole router. The caches of each processor tile are connected both to each other and to the system memory to provide overall system cache-coherence. Additionally, high-performance I/O devices allow the mesh to use DDR2 memories, and to be connected to 10-gigabit Ethernet and PCI-Express networks. Combined, these features make the TILE64 architecture particularly well-suited to processing high-speed sources of streaming data such as data-centre network routing.

- Centip3De designed by Fick, Dreslinski, Giridhar, *et al.* [16] at the University of Michigan combines 64 ARM Cortex-M3 processors into a 130nm stacked-3D package. Internally, the chip utilises through-silicon vias (TSVs) to communicate vertically between layers comprised solely of Cortex-M3 processors and layers comprised of 'cache-clusters' shared by up to four processors. A complete system may consist of four processor/cache layers (for up to 128) processors and up to three SDRAM layers (to a total of 256MB) Using near-threshold voltages (650mV–1.15V) and comparatively slow clock frequencies (10-80MHz) allow the devices to achieve 3,930 Dhrystone MIPS/W [17].

- Intel produced a prototype 'data-center-on-a-die' as a response to the trend of increasing core counts [18]. The chip embeds 24 pairs Pentium-class IA-32 processors in a 2D 6×4 grid network. Each pair of processors share a 'message-passing

buffer' which works with the grid network to realise of a message-passing protocol that is primarily realised by hardware. A specific 'message-passing memory type' presents this protocol at the software-level, whilst the caches implement the communication at the hardware-level. Coherency between all the processors in the chip is maintained using the same system. Linux can be booted on each core as they each implement the IA-32 instruction set.

- Orcale's SPARC M7 [19] is similarly designed for the data-centre and includes 32 S4 cores capable of supporting 256 threads simultaneously. Each core has exclusive L1 instruction and data caches, but L2 caches are shared by four processors in a SPARC cache cluster (SCC). Within this cluster, a single instruction cache serves all four cores, and two data caches serve two pairs of cores. The SCC tiles are then embedded in the on-chip network which connects to various coherency systems, four memory controllers, and eight 'database analytic accelerator' engines. Overall, the chip contains 64MB of L3 cache partitioned eight ways to give each SCC an equal share, and has a total off-chip bandwidth of 1TB/s for memory, multi-socket coherency, and I/O interfaces.

1.1 Performance Obstacles

1.1.1 Memory Bottlenecks

Clock frequency is not the sole driving force behind increases in single-threaded performance. The persistent requirement for higher density memory chips typically drives advances in fabrication technology because the number of transistors on a memory chip is directly proportional to the amount of data it can store. However, such heavily populated chips often suffer from poor parasitic traits which cause the operating frequency of memory devices to lag behind that of same-generation processors. Additionally, high-density memories usually employ dynamic logic which further increases data density at the cost of increased read and write latency.

Modern processors use a multi-layered approach to mitigate this introduced latency. Processors normally access dynamic system memories through at least one layer of cache. Modern desktop processors have have two or three levels of cache of increasing size and latency, and decreasing bandwidth. L1 caches are physically closest to the processor, usually operate on the same clock frequency, and are capable of presenting the processor with data one-byte-at-a-time if requested, thus shielding the processor from the 'bursty' nature of system memory. Caches farther away from the processor and closer to system memory become more tolerant of bursts of data, effectively shielding the latency of subsequent reads from the processor.

As a final defence against the comparatively high latencies of system memories, most systems use a *prefetcher* to perform *speculative loads* of addresses based on memory access patterns. If memory is being accessed in a predictable manner, system prefetchers will read additional data from the system memory so that it is already present in an appropriate level of cache *before* theprocessor requests it. Predictable accesses grant the prefetcher a sustainable lead on the processor and significantly reduce memory access times. Figure 1.2 shows how effective prefetchers can be when memory access patterns are predictable. Sutter [20] measured the data cache latencies for forward linear, reverse linear, pseudo-random, and random memory access patterns and clearly demonstrates that predictable accesses yield enormous performance increases over random accesses.

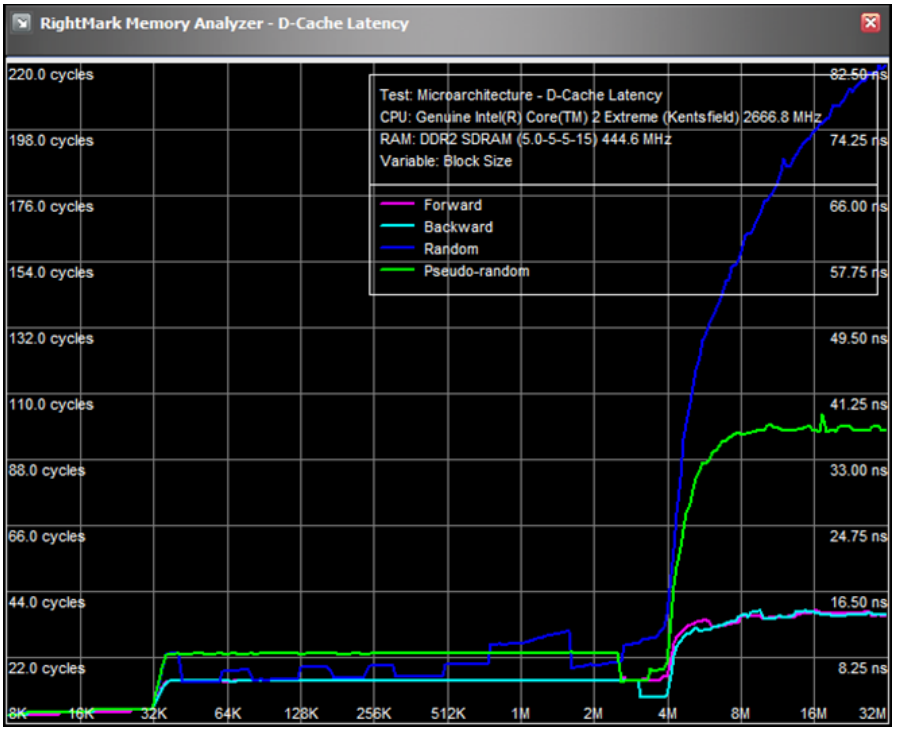


Figure 1.2: Data cache latencies for various memory access patterns on various amounts of data (figure produced by Sutter [20]).

Despite all of these techniques, Figure 1.2 still clearly shows that modern processors can consume data at a higher rate than it can be delivered. L1 caches and tightly-coupled memories are capable of delivering data in a small number of cycles, but large amounts of data causes high latency even when playing to the strengths of this multi-layered defence.

1.1.2 Instruction-level Parallelism

Many modern processors are capable of issuing several instructions simultaneously if they are independent and there is sufficient hardware to support them. As an example, consider equation (a) of Figure 1.3. Clearly the sum of A and $B+C$ cannot be computed

until $B + C$ is known. However, the value of A can be read into the register file whilst $B + C$ is being computed. A multiple-issue processor would be capable of performing these parallel tasks to gain a performance boost. Some processors take this concept further and include multiple functional blocks of the same type. Such a processor would be capable of computing (b) in two cycles by computing $A+B$ and $C+D$ simultaneously, and then computing the sum of the results on the following cycle.

(a) $Q = A + B + C$ (b) $Q = A + B + C + D$

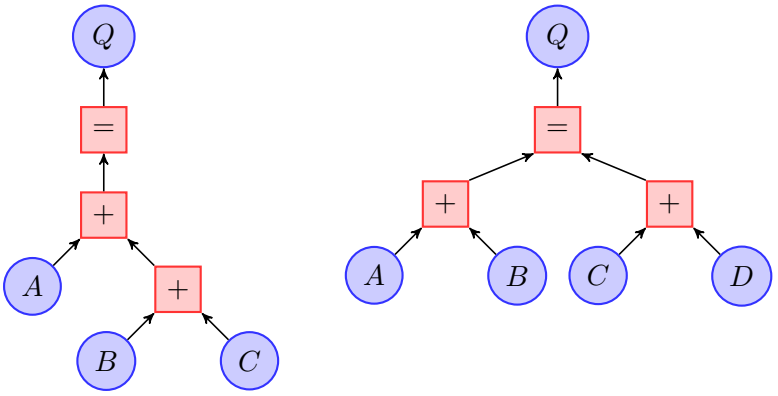


Figure 1.3: Two simple equations and the corresponding data-dependency graphs.

Static multiple issue is the simplest form of this technique, requiring that the compiler determines these potential optimisations *a priori*. The compiler constructs data-dependency graphs, such as in Figure 1.3, to determine exactly which instructions can be executed in parallel. Several instructions are then combined into a 'bundle' or an 'issue packet', more commonly known as a Very Long Instruction Word (VLIW), which is guaranteed by hardware to be dispatched simultaneously. Obviously, bundled instructions are subject to the constraints of the processor and two integer operations may only be executed in parallel if there are two integer arithmetic units present.

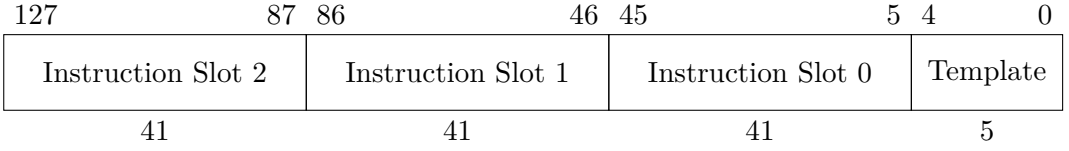


Figure 1.4: Intel Itanium instruction bundle format [21].

The Intel Itanium [21] is a mainstream VLIW processor which serves as a good demonstrator of how a VLIW architecture can dispatch instructions. Figure 1.4 shows that its VLIW format (called an *instruction bundle* in the Itanium nomenclature) is comprised of three instructions to be executed in parallel and then a *template* field providing some context for these instructions. Table 1.1 gives a small subset of the template codes used by the Itanium architecture, which map the instructions onto the various units within the processor. In addition to this mapping, the Itanium can enforce 'architectural stops'

Template	Slot 0	Slot 0	Slot 2
0x00	M-unit	I-unit	I-unit
0x01	M-unit	I-unit	I-unit
0x02	M-unit	I-unit	I-unit
0x03	M-unit	I-unit	I-unit
0x04	M-unit	L-unit	X-unit
0x05	M-unit	L-unit	X-unit
0x08	M-unit	M-unit	I-unit
0x09	M-unit	M-unit	I-unit

Table 1.1: First eight template instruction-mapping codes for the Intel Itanium architecture [21]. M-units primarily handle memory operations, I-units handle integer operations, and X-units handle extended length instructions of the form L+X.

(indicated by underlines in the table) if there are resource dependencies between subsequent instructions.

Dynamic multiple issue (more commonly known as *superscalar*) processors perform these optimisations at run-time rather than at compile-time. Clearly this requires more complex hardware as data-dependency must be detected as the instructions are presented for issue, but it carries a significant advantage over the VLIW approach: the hardware guarantees the correctness of the execution. VLIW processors will blindly execute the instructions as demanded by the compiler, so small architectural changes between hardware revisions can lead to performance degradations that can only be fixed by recompiling the software. Superscalar processors detect parallelism opportunities at run-time and are hence guaranteed to be correct.

Intel Hyper-Threading extends the superscalar concept by sharing function blocks between two copies of the architectural state [22]. This has the effect of each processor appearing as two 'logical processors' which both dispatch micro-operations to the shared compute resources. In addition to having a complete set of registers (including general-purpose) and a programmable interrupt controller, each copy of state also includes a 'return stack predictor' and an instruction translation look-aside buffer. Essentially, each copy appears as a complete processor from the software perspective.

Dynamic multiple-issue techniques are often further improved by allowing the processor to execute instructions in an order of its choosing. Out-of-order execution is common on larger processors and makes use of the data-dependency graphs already established by the multiple-issue hardware. As long as the processor can commit the results in the correct order—i.e., that of the input instruction stream—the end-result of the program is still correct. Consider again equation (a) of Figure 1.3, the overall sum cannot be computed until the result of $B + C$ is known, but another instruction disjoint to this equation can be executed in parallel.

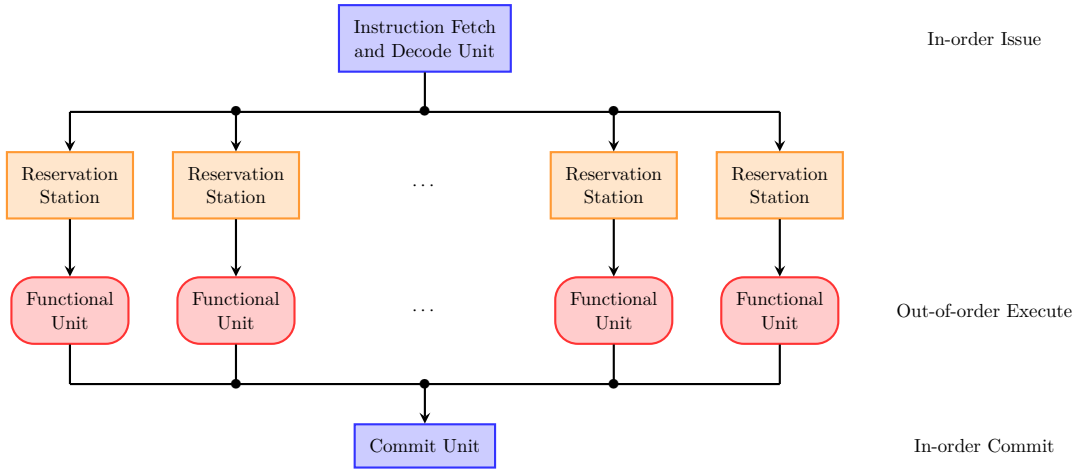


Figure 1.5: Three stages of an out-of-order execution pipeline, based on Figure 4.72 of Patterson and Hennessy [23].

Figure 1.5 shows the three stages of an out-of-order execution pipeline. Instructions flow into the fetch and decode unit in the order generated by the synthesis stage of the compiler. Operands and instructions are then dispatched to multiple functional units via their corresponding *reservation stations* which are *queues* that wait until all operands for an operation are available. Results from the functional units are then collated by the *commit unit* which both ensures the results are committed to the register file in the correct order, and distributes the results to the reservation stations waiting for operands. Continuing the earlier example: the commit unit would feed the result of $B + C$ back to whichever reservation station has $A + (B + C)$ queued.

1.1.3 Power

Larger transistor counts naturally lead to an increased power requirement for modern processors. Austin, Blaauw, Mudge, *et al.* [24] present equations (1.1)–(1.5) to illustrate how processor design parameters affect the overall power dissipation. Of particular interest is equation (1.1) which shows two factors contributing to overall power dissipation:

$$P = \underbrace{ACV^2f}_{\text{dynamic}} + \underbrace{VI_{\text{leak}}}_{\text{static}} \tag{1.1}$$

Dynamic power dissipation occurs while transistors are switching and is caused by the charging and discharging of the capacitive loads in the design. A is the proportion of gates actively switching and C is the total capacitive load of all gates. Clearly C is proportional to the number of transistors in the design, which is increasing with each new fabrication technology as explained earlier. Frequency, f, has largely plateaued in the

low-GHz range (i.e., $O(10^9)$). This leaves voltage, V, and A as two of the parameters that can be exploited to reduce dynamic power. However, the voltage cannot be scaled too low otherwise the clock frequency will drop correspondingly, as shown by equation (1.2)[1].

$$f_{\max} \propto \frac{(V - V_{\text{th}})^{\alpha}}{V} \tag{1.2}$$

Static power is primarily described by the leakage current, I_{leak}, in equations (1.3)–(1.5). There are two contributions here: the *sub-threshold leakage current*, I_{sub}, and the gate-oxide leakage current, I_{ox}. Across both of these equations, the parameters K_1, K_2, n, β must be determined experimentally for each fabrication technology level, W is the transistor gate width, V_θ is the thermal voltage, and T_{ox} is the gate oxide thickness. V_θ is typically 25mV at room temperature, but increases linearly as temperature increases.

$$I_{\text{leak}} = I_{\text{sub}} + I_{\text{ox}} \tag{1.3}$$

$$I_{\text{sub}} = K_1 W \exp\left(-\frac{V_{\text{th}}}{nV_\theta}\right) \cdot \left[1 - \exp\left(-\frac{V}{V_\theta}\right)\right] \tag{1.4}$$

$$I_{\text{ox}} = K_2 W \left(\frac{V}{T_{\text{ox}}}\right)^2 \cdot \exp\left(-\frac{\beta T_{\text{ox}}}{V}\right) \tag{1.5}$$

Austin, Blaauw, Mudge, *et al.* [24] note that equation (1.4) suggests two ways to reduce static power: firstly, to turn off the supply voltage, V, as often as possible, and secondly to increase the threshold voltage, V_{th}. However, the latter would cause lower operating frequencies due to equation (1.2) just as reducing the supply voltage would. Austin et. al also note that while increasing the oxide thickness, T_{ox}, in equation (1.5) is the obvious way to reduce the leakage current contribution, it is also not viable because oxide thickness must decrease with device scaling.

Increased power dissipation naturally requires improved cooling, which itself becomes complicated by the shrinking die-size of modern chips. The heat densities involved are high enough to make efficient cooling a difficult problem. Power must therefore be reduced inside the device by dynamically scaling the voltage and frequencies at run-time, or by power-gating large parts of the chips so that unused resources are simply not drawing any power.

Esmaeilzadeh, Blem, St. Amant, *et al.* [25] constructed a model of power dissipation for CPU-like multi-core architectures (in contrast to GPU-like) and analysed the effects of the aggressive scaling given by ITRS[2] projections. The authors determined that 8nm fabrication technologies would lead to over 50% of a chip being underutilised due to power constraints.

[1] $\alpha \in [1, 2]$ and is usually around 1.3, but specific values must be determined experimentally.
[2] International Technology Roadmap for Semiconductors (http://www.itrs.net).

1.1.4 Reliability

As devices become smaller and their numbers increase, reliability becomes a significant issue. Chien and Karamcheti [26] look at the trends in flash memory chips (notoriously focused towards high-density integration) and note that while the density has been steadily increasing over the past decade, the durability of the devices has not. Additionally, read and write times have actually been *increasing* whilst the endurance of the devices has been decreasing. They consider this to be the "'first ending" of Moore's law' and that flash memory is merely the first victim.

Flash memories are typically not subject to the same high frequencies (and hence high dynamic power dissipation) of processing logic. The previous section showed how smaller and faster devices are now leading to higher power dissipation. Kish [27] analyses runtime reliability by studying how thermal noise can cause spurious bit-flips. However, these can be eliminated by keeping the threshold voltage above a certain minimum governed by the fabrication technology. Clearly this compounds with issues presented previously on power, and strengthens the case for an increased threshold voltage. However, this imposes restrictions on maximum clock frequency in accordance with equation (1.2).

1.1.5 When Taken Collectively

Looking at the entire single-threaded computing picture, the issues discussed in this section show that it is clearly reaching its limit. Single-threaded performance increases have been brought about by embracing parallelism within the architecture itself (e.g., superscalars, hyper-threading, and out-of-order execution). Conventional performance increases from merely increasing the clock frequency and/or shrinking the component sizes are simply not as effective with modern fabrication technologies.

Power dissipation and device fabrication reliability favours larger devices being clocked more slowly, but this is not conducive to an improvement in single-threaded performance. With modern processors including greater numbers of cores, and technologies like hyper-threading presenting the software layer with *logical cores*, the clearest path forward is to investigate parallel computation at the program level.

1.2 Parallel Software Structure

Many programs are typically single-threaded, which clearly does not make good use of modern computing hardware. Despite this, some software (such as video games and scientific software) will make use of graphics cards as *stream processors* to significantly improve the performance of calculations. Other systems employ varying levels of heterogeneity so that code can be executed on the most power-efficient compute resource that

can achieve the desired outcome [28]. ARM's big.LITTLE [29] technology serves as a good example that combines a small but highly power efficient processor (e.g., Cortex-A7) with a high-performance processor (e.g., Cortex-A15). For simple applications, the higher-power processor can be entirely disabled, only needing to be powered up when the application requires greater capability.

Traditional multi-threaded programs that make use of multiple homogeneous processors employ synchronisation primitives such as atomic memory accesses, and locks/mutexes/semaphores to protect resources. Programming with these primitives is notoriously difficult, with limited debugging capability and increased opportunities to introduce deadlock. Incorrectly ordered locking or poorly written locking code can cause programs to stall in an entirely unrecoverable manner. To compound matters, these synchronisation primitives are also used for inter-thread communication as well as access to physical resources.

On the other end of the computing spectrum, micro-controllers afford programmers the freedom to carefully choreograph the program flow to prevent ill effects. With direct access to registers and memory-mapped system resources, programmers are free to perform cycle-level optimisations and interleave expensive tasks with cheaper on-going ones. A good example of this is the V-USB[3] library for use on Atmel AVR micro-controllers. The library implements the USB standard, and uses carefully written assembly code to interleave signal synchronisation and decoding code with USB waveform generation. However, the increased complexity and capability of more sophisticated systems drastically complicates this to the point of making it impractical outside of micro-controller systems.

1.2.1 Scalability

Software scalability is central to taking advantage of an increasing number of readily available processors. If workloads cannot be effectively distributed over these many cores, then there can be little observable speed increase.

Early parallel machines were considered *vector* machines because they would apply a single operation to multiple pieces of data simultaneously. This technique became known as Single Instruction Multiple Data (SIMD) and will be explained in Chapter 2. However, parallelism may take many guises such as the now common *symmetric multi-core* approach in modern desktop computers, where identical cores are replicated several times.

In 1967, Gene Amdahl compared three systems of varying levels of parallelism and concluded that processors with greater sequential performance than parallel performance

[3] https://www.obdev.at/products/vusb/index.html

lead to an overall performance increase when applied to parallel workloads [30]. This seems rather counter-intuitive, as the most parallel architecture that Amdahl considered was a SIMD machine containing 32 arithmetic units, and the least parallel architecture was a modified vector processor capable of issuing individual operations at the same rate as vector operations.

These results imply that parallel performance increases are limited by the ability to process sequential parts quickly. Consider the SIMD machine performing house-keeping tasks between computation tasks; in the worst case, the house-keeping tasks might require iteration whereby most of the SIMD machine remains idle. This observation is known as Amdahl's Law as in equation (1.6),

$$S = \frac{1}{s + \frac{p}{N}}, \tag{1.6}$$

where s is the fraction of sequential tasks in a workload, p is the fraction of parallelisable tasks, N is the number of processors performing parallel work, and S is the total speed-up ratio achieved. A clear, and concerning, result of equation (1.6) is that even with an infinite amount of parallel processing capacity, the maximum achievable performance increase is limited to $1/s$.

However, this presumes that the workload in question was originally sequential and a performance increase is sought by splitting parts of it out over multiple cores in parallel. An alternative viewpoint is to look at how long a parallel task would take to complete if it was made sequential. Gustafson [31] states that 'when given a more powerful processor, the problem generally expands to make use of the increased facilities.' Thereby implying that looking at performance *increases* obtained from making a sequential task parallel is not the correct perspective. Gustafson's alternative speed-up equation (often called the Gustafson-Barsis law) is given as equation (1.7).

$$S = \frac{s + pN}{s + p} \tag{1.7}$$

$$= s + pN \tag{1.8}$$

$$= N + (1 - N) \cdot s \tag{1.9}$$

Amdahl's analysis assumed that a workload of of $s + p = 1$ on a sequential computer would become $s+p/N$ on a parallel computer, whereas Gustafson's began with a parallel compute workload of $s + p = 1$ and converted it into a sequential workload of $s + pN$. The key result here is that workloads should be *designed to be parallel right from the start.*

Hill and Marty [32] extend these insights further and find that for small core-counts, simple elements with high sequential performance are more favourable for highly parallel workloads. For workloads with a roughly equal share between sequential and parallel fractions, techniques such as hyper-threading are more favourable. Asymmetric systems tended to perform well with high numbers of highly-capable cores.

1.2.2 Modelling

Concurrent systems are notoriously difficult to model as well as program. Various model-checking techniques and languages exist to verify that the desired behaviour is itself correct before any production code is written to implement the behaviour.

The *actor* model was originally proposed as a method of writing programs that are correct by treating each requirement of a program as a disjoint unit [33]. These units may only interact by sending messages to each other. Actor-based programs are therefore simple to parallelise, in principle, because the data-dependency has been explicitly expressed in the way that messages are sent.

Interestingly, with the actor model, the sequential work performed by each actor is the processing of its inbound message queue and the transmission of consequent messages, thus minimising the sequential fraction of work and playing to the strength of Amdahl's law.

An alternative approach is to consider concurrency from the outset. Communicating Sequential Processes (CSP) treats *concurrency*, *input*, and *output* as primitive operations and supports the idea of composable software [34]. CSP structures software based on the flow of data between multiple stages until the desired result is achieved. CSP is a language as much as a software architecture, and is capable of describing problems that may cause processes to enter and leave existence as required. Despite being a general technique, Hoare acknowledges that 'it would be unjustified to conclude that all these primitives can wholly replace the other concepts in a programming language.'

Both of these techniques are capable of expressing the more commonly known synchronisation primitives (such as mutexes and semaphores), but this makes them susceptible to issues such as deadlock. Ultimately, these techniques (as with many others) do not completely remove the difficulty of writing parallel software, but instead limit the areas of a program that many contain these obstacles and errors.

1.3 A New Perspective on Software

To properly leverage parallelism will require changes in how both hardware and software are constructed. This chapter has given examples of how power dissipation, memory

latency, material characteristics, and reliability are negatively effecting further improvements to single-threaded performance. Unfortunately, these issues also impact current approaches used for parallel hardware.

Unconventional architectures are likely to bring increased parallel performance, but will require a new perspective on how to craft software. To address the issues presented in this chapter, these architectures (both hardware and software) will need to:

1. Employ methods to further minimise memory latency to ensure that sequential performance is not slowed by processors being starved of data.
2. Minimise the sequential fraction of programs, ideally limiting it entirely to handling communication. By focusing on work that can be performed in parallel, software will be less constrained by the limit of Amdahl's equation.
3. Minimise the cost of sharing resources. This point also meshes with the previous when some of the shared resources are those used for communication. Reducing the costs introduced by synchronisation primitives used to protect these resources has far-reaching benefits.
4. Fundamentally support parallelism rather than treating it as a feature to provide. Parallelism must be approached from a Gustafson perspective, where an equivalent sequential approach is clearly much slower, rather than from an Amdahl perspective where the parallel approach is an afterthought.

1.3.1 SpiNNaker—A Neuromorphic Computing Platform

SpiNNaker is a neuromorphic computing engine designed to rapidly accelerate neural simulations [35]. A key design intent of the machine is to support neural simulations containing up to around 10^9 neurons in *biological real-time*. It achieves this by embedding a total of 10^6 conventional ARM processors in a novel biologically-inspired routing fabric optimised for sending very small packets of data with very small latencies. Crucially, it addresses the requirements listed in the previous section despite occupying a different domain:

1. Each of the cores have small tightly-coupled memories (TCMs) for instructions and data, which inherently minimise access times. Whilst a system SDRAM is also present, it may not contain instructions and the data resident within it should be—ideally—less-frequently required. To mitigate the much higher latency of the system SDRAM when compared to the TCMs, each core is equipped with a DMA controller which can be used to mask the delay.

2. Communication between the cores is brokered entirely by hardware, and uses interrupts to finally deliver the messages to the target cores [6]. This drastically minimises the communication cost on the cores themselves, leading to low sequential workload contribution.

3. The hardware-brokering of messages extends to the other shared hardware resources of the system too. *All* hardware resources are arbitrated by hardware, and all low-level synchronisation primitives are implemented in hardware. Despite this, the system software must still determine by a sequential search which neuron models must be updated by message arrivals. However, this contribution is small in comparison to the processor time allotted to neuron model execution.

4. Finally, SpiNNaker and the software simulation model that it employs have been co-designed from the ground up to be parallel. The target application is the parallel simulation of neural networks, and the hardware has been designed to support that.

In addition to these remarks, even a small SpiNNaker machine consisting of a single board contains almost one thousand processors, which facilitates investigations into massively-parallel computing in an extremely convenient environment.

1.4 Contributions

SpiNNaker provides massive parallelism to a very specific domain where it performs very well. However, the primary building-blocks are conventional ARM processors with high sequential performance. The neuromorphic design criteria restrict the problem-sets for which SpiNNaker is suited, but they do not preclude non-neural problems from being attacked.

This is the specific problem that this work addresses—to run non-neural applications on this neuromorphic platform to exploit its massive parallelism for other problem domains. The key contributions resulting from this work are as follows:

Firstly, SpiNNaker was designed with the resilience of neural systems in mind, which allows relaxed reliability constraints in certain circumstances. Transient (or permanent) hardware faults that limit network traffic can be largely ignored by neural applications because significant loss of communication can be inherently tolerated. Biological systems are intrinsically noisy and lossy, so provided that any inaccuracies caused by shortcomings of the simulation platform are less than the biologically realistic losses, any effects can be ignored because they do not *significantly* affect the overall accuracy of the simulation.

The first contribution addresses this reliability requirement by presenting a set of algorithms that can discover any faults in the system, accurately configure the networking resources in a decentralised manner, establish a control tree that can be used for more traditional parallel- processing behaviours where appropriate, and present all of this information for download so that external tools can accurately map problems onto the *discovered* machine topology. This work has been published as Dugan, Reeve, and Brown [3] and Dugan, Reeve, Brown, *et al.* [4].

Secondly, a simple demonstrator application is developed to show how non-neural simulation software can be structured to take advantage of the machine architecture, thus showing that simple processors with small memories are highly scalable with appropriately structured software. This work as been published as Brown, Mills, Reeve, *et al.* [1], Brown, Mills, Dugan, *et al.* [5], and Brown, Mills, Dugan, *et al.* [7].

1.5 Organisation

Figure 1.6 shows the tool-chain that is built up over the course of the thesis.

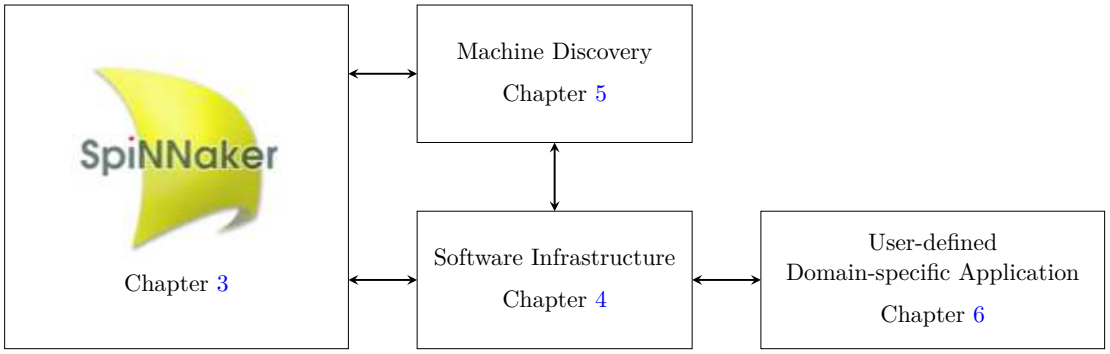


Figure 1.6: High-level view of the tool-chain developed in the thesis, identifying how each chapter inter-relates.

The thesis is structured as follows: Chapter 2 introduces the general concepts of parallel computing and provides a survey of the neuromorphic computing landscape. A detailed description of SpiNNaker is given in Chapter 3. Chapter 4 introduces the tool-chain and system configuration process that supports the subsequent chapters. Chapter 5 presents a set of decentralised algorithms that discover the structure of an unknown SpiNNaker machine, configure the routing hardware, and packages this data for download. Chapter 6 presents an application that both demonstrates the tool-chain from a user perspective and illustrates how to structure non-neural applications for use on SpiNNaker. The reliability of the hardware and software presented in the prior chapters is discussed in detail in Chapter 7. Chapter 8 reviews the work presented in this thesis, discusses potential directions for future work, and closes by reflecting on the contributions made.

Chapter 2

State of the Art

The goal of this chapter is to familiarise the reader with the techniques employed in conventional parallel architectures, before re-framing them in a neuromorphic context. We begin by looking at the structure of conventional parallel software and the inherent difficulties and challenges associated with it. This naturally leads onto a discussion of communication techniques, which address perhaps the most significant difficulties in parallel implementations. Finally, a survey of various software libraries and frameworks is presented to provide examples of real-world parallel programming.

In this instance, neuromorphic systems are a special case of parallel systems. A brief survey of the current neuromorphic hardware landscape is presented to illustrate the differences between them and the conventional systems previously discussed. Next, a survey of the software landscape is given, finally leading onto a discussion of neuromorphic description languages.

In the final section, SpiNNaker, as the chosen platform for the work presented in this thesis, is introduced as a machine capable of bridging the gap between the conventional and neuromorphic parallel computing domains.

2.1 Conventional Parallel Computing

2.1.1 Hardware Taxonomies

Software (and hardware) can be broadly categorised by how the streams of instructions and data flow through the processors. Table 2.1 shows what is commonly known as *Flynn's Taxonomy* [36]. Low-level architectural features that enable instruction-level parallelism (such as superscalar and out-of-order execution) do not affect the classification because they cannot be sensibly reasoned about in a globally parallel setting [37].

	Single Data-stream	**Multiple Data-streams**
Single Instruction-stream	SISD	SIMD
Multiple Instruction-stream	MISD	MIMD

Table 2.1: Flynn's computer organisation taxonomies.

In addition to the classifications in Table 2.1, Single-Program Multiple-Data (SPMD) and Multiple-Program Multiple-Data (MPMD) are commonly used to characterise the nature of the instruction stream as well as the architecture. As explained later, MIMD implies synchrony between individual instructions even though they may contain different op-codes. However, SPMD and MPMD permit instruction streams to operate at different rates across the machine.

2.1.1.1 Single-Instruction Single-Data

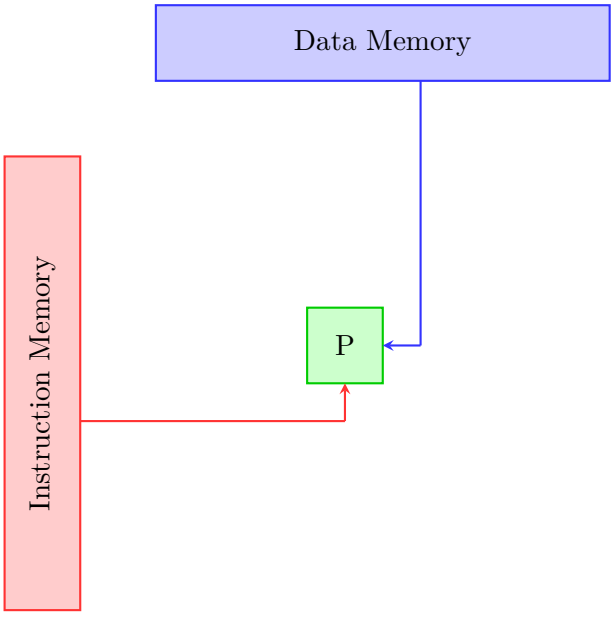


Figure 2.1: Single-Instruction Single-Data (SISD) architecture.

Of all of these taxonomies, SISD is the most common because it represents a conventional single-threaded computer. Instructions and data are both drawn from dedicated stores as shown in Figure 2.1. Most applications ranging from desktop applications down to small embedded micro-controller applications fall into this category.

2.1.1.2 Single-Instruction Multiple-Data

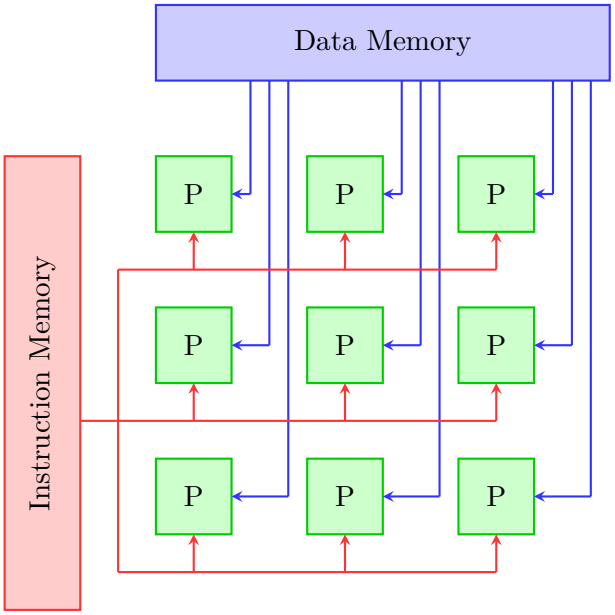


Figure 2.2: Single-Instruction Multiple-Data (SIMD) architecture.

In a SIMD architecture, instructions are shared between all processors but the data each is acting upon is specific as shown in Figure 2.2. These are often called *vector processors* or *vector operations* because each instruction leads to a set of results rather than just a single one. It is important to note that vector operations cannot reduce a set of operands into a single result, such as computing the sum of all elements, x_i, of a vector, $\vec{X}$. However, consider instead $\vec{Z} + \vec{X} + \vec{Y}$ being expressed as $z_i = x_i + y_i$, in this case, a SIMD computer with k cores can compute up to k of these sums in parallel.

SIMD instructions are particularly well-suited to applications such as geometric calculations where the input and output data-structures are well-formed and regular. Computer graphics is an obvious and common problem that desktop computers face that is amenable to this sort of performance boost. Google have recently added SIMD support to their Dart language—designed for use in modern web-browsers—to benefit 3D graphics and other multimedia implementations [38].

In most architectures, SIMD instructions are used to complement an otherwise SISD workload. Even highly-parallel supercomputers such as the IBM Blue Gene/Q [39] and highly domain-specific computing engines such as the Anton Molecular Dynamics machine [40] take this approach. Despite the ratio being heavily skewed in favour of non-vector instructions, SIMD leads to drastic performance increases by performing particularly costly instructions in parallel. A suitable example of this is the vector square-root instruction, `SQRTPS`, in Intel's SSE instruction set extensions [41, pg. 4-379].

It should come as no surprise that careful use of SIMD instructions can yield significant performance increases in regular data-sets [**Cohen2003**]. However, irregular data-sets

can reportedly see speed-ups of up to 9 times that of a serial implementation if they can be mapped into an appropriate form [42]. Similarly, highly-relational database workloads can see large performance increases by using SIMD instructions in common operations such as selection scans and hash-key generation [43].

2.1.1.3 Multiple-Instruction Single-Data

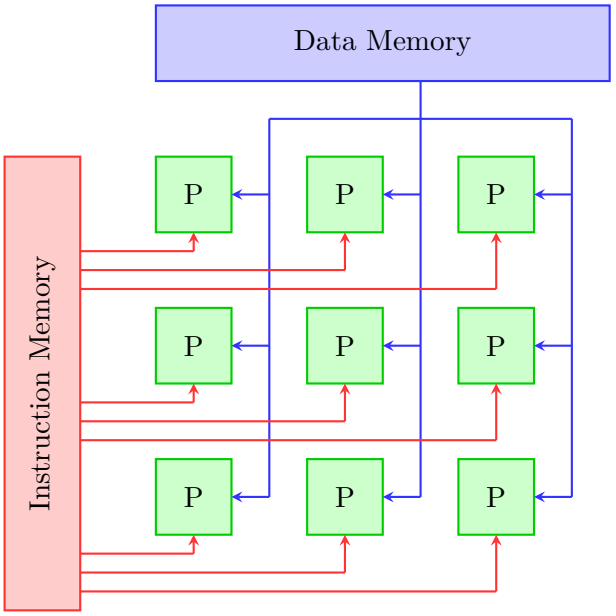


Figure 2.3: Multiple-Instruction Single-Data (MISD) architecture.

MISD is a seldom used architecture and tends to be focused on a specific problem when it is. The control flow of MISD allows for multiple different results for a single computation to be obtained simultaneously without a performance penalty when compared to an equivalent SISD approach. Effectively it enables the *speculative computation* of many results, but requires an arbitration step afterwards to choose the correct result. Halaas, Svingen, Nedland, *et al.* [44] show that this approach can be used for matching patterns in a data-stream; multiple processing elements search for patterns in the data-stream simultaneously, and the results of this are fed into a tree of result-selection nodes which selects the most appropriate result.

Alternatively, a problem that requires several disjoint computations before a final result can be determined can benefit. The M-Buffer architecture [45] uses MISD for Z-buffer computation when composing a graphics frame for final display to the user.

2.1.1.4 Multiple-Instruction Multiple-Data

MIMD is a particularly broad classification that loosely follows the architecture shown in Figure 2.4. An architecture with data and instruction memories distributed across

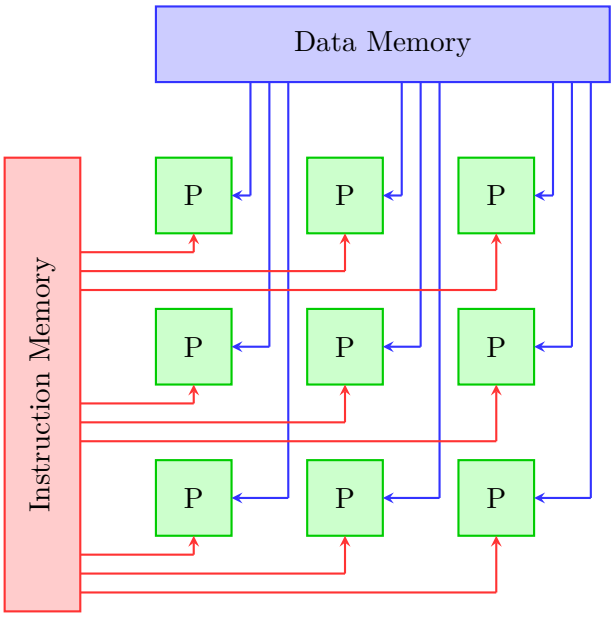


Figure 2.4: Multiple-Instruction Multiple-Data (MIMD) architecture.

many physical machines may still be considered MIMD. For example, desktop multi-core processors fall into this classification. However, the term is often specifically reserved for architectures that maintain cache-coherence with more specific classifications (described next) for multi-machine behaviours.

Cache-coherency is of particular importance in MIMD architectures and will be explained in more detail in the next section. Briefly, it refers to ensuring that all processors sharing a memory have the same value stored in their caches. Consider the case of two processors, P_0 and P_1, both performing a calculation involving some variable, A. Both processors have A loaded into their data-caches, but P_0 has just performed an operation that alters A. When P_0 commits this new value to its own cache, it must also be committed to the cache of P_1 to ensure that all calculations thereafter remain correct.

Strategies employed to maintain cache-coherency vary between architectures. TILE64 [15] uses an L2 cache shared between several neighbouring tiles to unify the contents of the processor-local L1 caches. Centip3De [17] places caches that are shared locally between four processors on the system network, which support specific coherence traffic. Intel's prototype array processor [18] includes a hardware-supported message-passing protocol to maintain cache-coherency. The SPARC M7 [19] uses a specific network to maintain coherency both within a single chip and within a multi-chip cluster. As a final example, the IBM Blue Gene/Q chip [39] uses L2 caches to maintain coherency similarly to TILE64, but also extends this capability off-chip in a similar manner to the SPARC M7.

2.1.1.5 Single-Program Multiple-Data

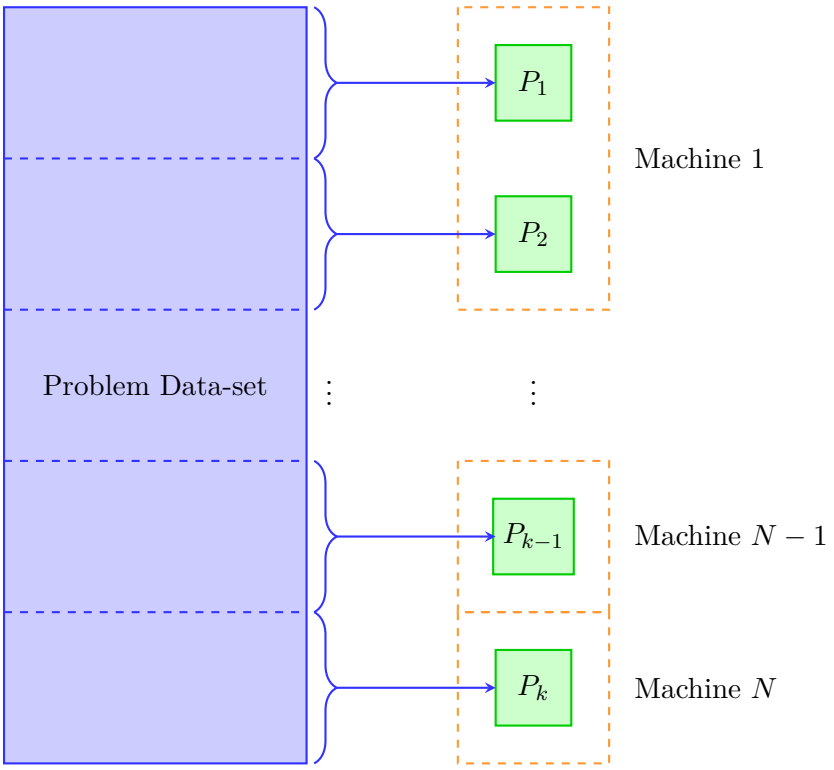


Figure 2.5: Single-Program Multiple-Data (SPMD) architecture.

SPMD is a subset of MIMD where there is no implication of instruction-level synchrony; each program is allowed to execute at its own pace. *Cluster computers* are the most common example of this, where they are essentially a large number of networked processors that make no attempt to maintain cache-coherency. *Sequence points* may be introduced into the control flow at regular points to enforce a level of global synchrony if required. For example, if an application is comprised of a sequence of steps that each depend on the result of the previous.

A *barrier* is a common way of introducing sequence points. Consider the set-up shown in Figure 2.5. Suppose a workload is uniformly distributed over processors $\{P_i; i \in [1, k]\}$ in a roughly uniform manner, but, for the sake of argument, P_1 and P_2 are slower than the others. All processors begin working on their specific task and then enter a barrier state once they are complete. A processor may not then leave the barrier until *released*, which is usually issued once all the processors have entered the barrier state. The behaviour resulting from this is that all processors advance at their own rate towards a specific goal, and may not begin working towards the subsequent goal until *all* processors have finished.

Typically, an SPMD approach is used to solve problems at a high level of granularity with each section assigned to a dedicated processor as shown in Figure 2.5. However,

Intel created an experimental compiler, `ispc`, which can take a conventional C++ program as input and synthesise a binary that can be distributed over many processors [46]. Homogeneous core sets are assumed to take advantage of any SIMD features available locally. Sequence points are automatically inserted to ensure the correctness of the results, which opens this technique up to fine-grain parallelism. Additionally, the compiler introduces the concept of 'coherent control flow' and 'uniform data-types' which increase performance, in certain cases, through near lock-step instruction execution.

2.1.1.6 Multiple-Program Multiple-Data

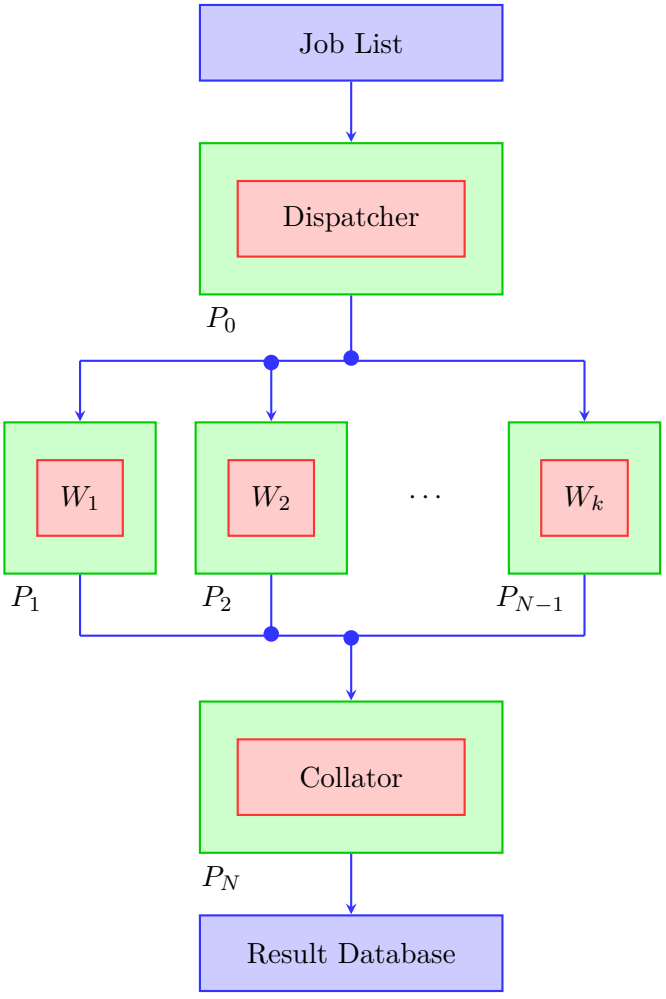


Figure 2.6: Multiple-Program Multiple-Data (MPMD) job-based architecture.

MPMD architectures are not as rigidly constrained as the others presented up to this point. Multiple instruction and data streams afford a high degree of flexibility to tailor the approach to the particular problem at hand. A typical architecture given in Figure 2.6 processes a unit of work called a *job* as they arrive. This is appropriate for workloads with a high degree of variance in the data or the processing of each job. In the example in Figure 2.6, the Dispatcher node is responsible for issuing a job to the most

appropriate Worker. There is no requirement for the workers to be homogeneous, so the "most appropriate" worker might be the one running the correct program to process the data of the job.

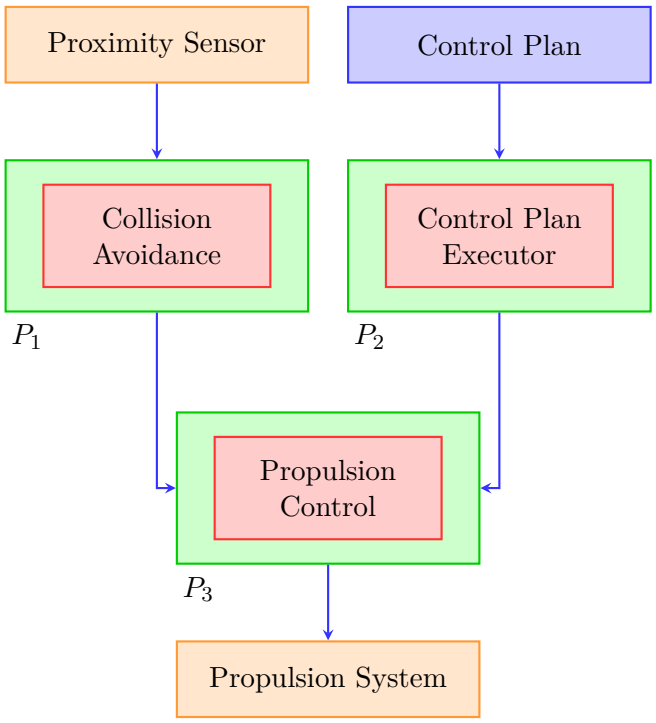


Figure 2.7: Multiple-Program Multiple-Data (MPMD) distributed architecture.

An example of an MPMD application can be seen in the robotic control application of Figure 2.7. In this case, each component of the robot has a program associated with it to perform specific processing. These programs may be running on a single processor or distributed across various processors in the robot, each potentially using a different operating system [47]. ROS[1] is an example of an open-source platform designed to support these kinds of architectures.

2.1.1.7 Non-uniform Memory Access

Shared memory computers, regardless of any other classification, fall into one of two categories. Systems in which each processor has a direct connection to the system memory have *Uniform Memory Access* (UMA) as shown by Figure 2.8 [23]. In these systems, each processor pays the same latency cost to access any part of the shared memory; all memory access times are uniform. Previously mentioned examples of this are TILE64 [15] and Centip3De [17].

[1] Robot Operating System—http://www.ros.org/

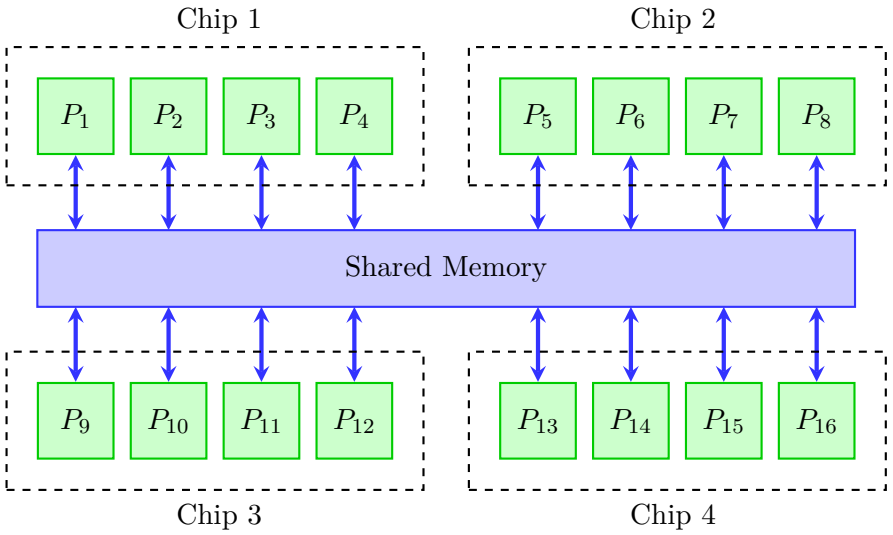


Figure 2.8: Uniform Memory Access (UMA).

However, memory devices will typically have insufficient ports to service a large number of processors at maximum efficiency. A *Non-uniform Memory Access* (NUMA) architecture, as shown in Figure 2.9, reduces the number of processors connected to a single memory, thus improving the memory bandwidth per processor.

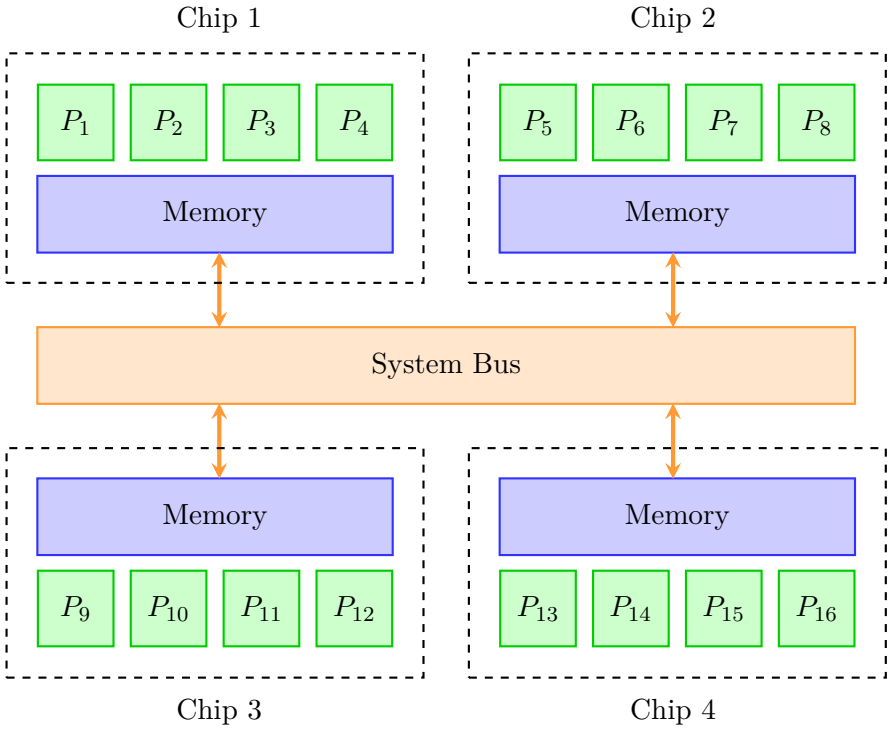


Figure 2.9: Non-uniform Memory Access (NUMA).

This technique is well suited to problems where each processor operates on a disjoint piece of the overall problem, because long-range memory accesses must now pass through the system bus which will typically be slower than the memory interface.

2.1.1.8 General-Purpose GPU (GPGPU)

Modern Graphics Processing Units (GPUs) are highly optimised SPMD architectures targeted and rendering advanced 3D graphics in as little wall-clock time as possible. They are fundamentally streaming devices with several processing stages capable of operating on many pixels in parallel, as shown in Figure 2.10.

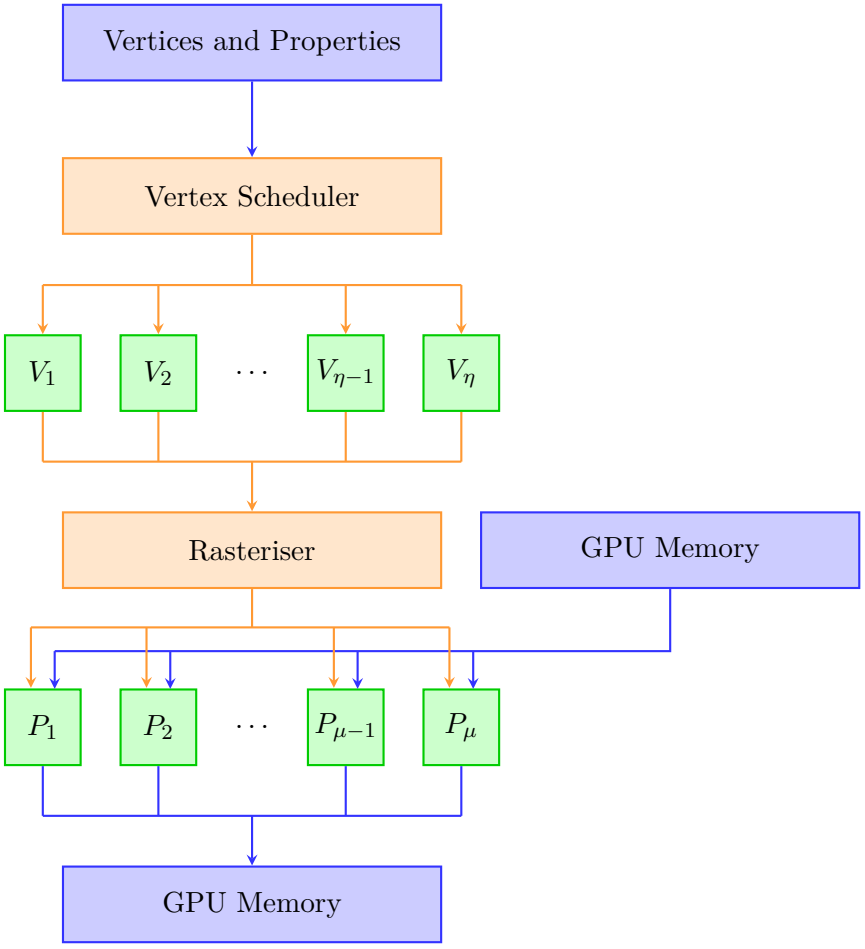


Figure 2.10: Simplified block-diagram for a generic GPU (adapted from Figure 1 of Tarditi, Puri, and Oglesby [48]) with η vertex processors and μ pixel processors.

The vertex- and pixel-shader processor stages each implement highly-parallel floating-point instructions to support the huge number of vector and matrix manipulations required in 3D graphics. By mapping non-graphical data-sets into the appropriate graphics memories, this enormous compute resource can be used to accelerate many other computations without requiring a large-scale MIMD machine [48].

An unfortunate weakness of this approach is the large overhead of setting up a GPU. Compiling shaders is usually performed at run-time so that the GPU hardware can be correctly targeted, and the input data-set must be appropriately organised to fit into

the GPU texture memory. GPGPU computation is therefore not universally applicable and has a set-up cost that must be mitigated before a performance gain is noticeable.

2.1.2 Difficulties

2.1.2.1 Resource Contention

It is rare that a program, of any sort, does not need to access some system resource in order to complete its task. These resources include particular in-memory data-structures, data-files on storage media, and specific hardware resources such as sensors or specific accelerators. Parallel computing complicates this issue because several concurrent processors may require access to a resource being shared between all processors on the host machine, and perhaps even across the whole network of a cluster machine.

Dijkstra [49] identified the parts of a program that access these resources as 'critical sections' and presents an algorithm to protect them, introducing the concept of "locks." The algorithm is predicated on the notion that no processor may enter its critical section until all processors are sure which one has been granted access. Once the work inside the critical section is complete, the processor releases the lock and the arbitration process begins again. These locks are commonly called *mutexes* which is a contraction of "mutual exclusion."

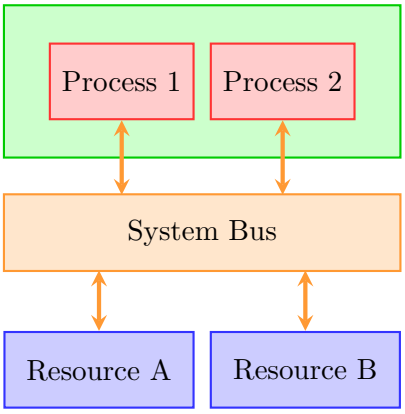


Figure 2.11: Example architecture containing a single processor, running two processes, connected to a bus hosting two system-level resources.

Consider the hardware architecture given in Figure 2.11. In this example, both software processes are executing on a single processor (though this is not required to maintain the correctness of the approach) and are planning to make use of Resource A. Associated with the resource is a mutex, M_A, to protect against contention. Figure 2.12 shows a flow of events that could occur during the execution of these processes. It is important to note here that (a) there is no guarantee that Process 1 would be granted access first on physical hardware, and (b) Process 2 stalls until Process 1 relinquishes the lock.

Enforcement of this software protocol ensures that *mutually exclusive access* is granted to Resource A.

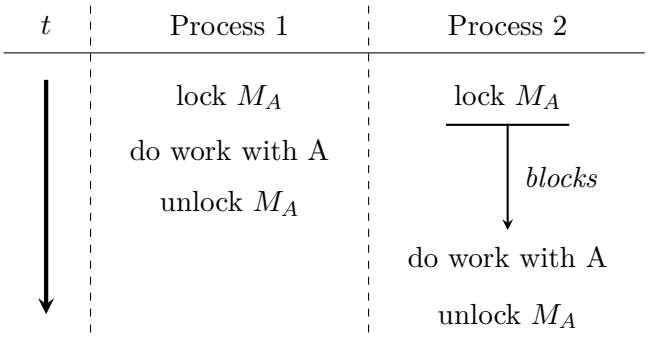


Figure 2.12: Two processors using a single lock, M_A, to protect a single resource.

Dijkstra's original formulation required that a single memory location served, essentially, as an arbiter, and that any write race-conditions were appropriately resolved by the memory hardware. Also assumed is that all processors in the system have uniform memory access privilege, which is likely to be true on an architecture such as that in Figure 2.8. Consider instead a NUMA architecture (Figure 2.9) where only chip-local processors have write-access, and all others are only granted read access: clearly, the algorithm will no longer work.

An advancement is to use the "bakery algorithm"[2] [50] in which all processors assign themselves a unique number locally, the lowest of which (not necessarily the processor that attempts to lock the mutex first) is granted access to their critical section. It is assumed that all processors have read access to all distributed memories. Processors mark their interest in the mutex before embarking on a hardware-mediated controlled race condition to assign the unique identifiers. Once all identifiers have been assigned, locks are granted in a distributed manner until all have been granted exclusive access once.

2.1.2.2 Deadlock

The example in Figure 2.12 is somewhat contrived because both processes are vying for a single resource. Consider instead the example in Figure 2.13 where both processes are vying for both resources. Process 1 locks M_A at the same time Process 2 locks M_B, next both processes attempt to acquire mutexes already held by the other process. The two processes are now in an mutually-assured *deadlock* from which neither can escape [51, pgs. 151–152].

Deadlock avoidance is a notoriously difficult parallel programming problem because of the issue shown in Figure 2.13, which is astonishingly simple to produce. A potential

[2]The arbitration method is similar to a bakery where each customer draws a ticket with a unique number which is called in order.

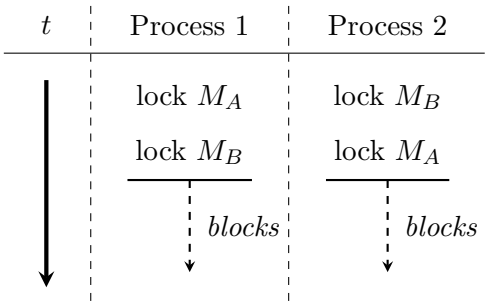


Figure 2.13: Two processors using a pair of locks, M_A and M_B, to protect the two resources of Figure 2.11.

workaround here would be to simply not use any locks at all. Software Transactional Memory (STM) replaces the notion of locks with a *transaction log*, local to each thread, containing all variable changes to be committed to main memory at the conclusion of a block of execution. By introducing constructs that guarantee a block only contains memory operations (i.e., none that are irreversible, such as commands issued to some peripheral device), transactions can be checked against the system memory, atomically, before being committed. If the target memory has been modified by an existing thread, the block of code can be executed again based on this new data and a new transaction log constructed. Embracing this technique allows parallel code to be written without resorting to locks, hence removing the potential for deadlock [52].

2.1.2.3 Synchronisation

Synchronisation between multiple threads of execution spanning multiple processors is of particular interest on SPMD and MPMD architectures. The problem to be solved will be divided into smaller parts, on which each concurrent program can work independently. Perhaps the most obvious case for synchronisation is merely detecting when all the parallel threads of execution have completed their task. Computations involving several independent stages, where each subsequent stage builds on the results of the previous, serves as another example.

A commonly employed technique to achieve this is *barrier synchronisation*. Each processor may *enter* the barrier whenever the implementation deems it appropriate (e.g., after a result has been computed), but may not leave the barrier state until it has been *released*. Typically the release is triggered by all processors having entered the barrier, thus allowing advancement into the next stage of processing.

Barriers may be implemented using shared or distributed memory, and often take advantage of the mutual exclusion locks previously described [53]. Alternatively, the barrier can be achieved by specialist hardware as in the IBM Blue Gene/Q [54]. In this particular implementation, the barrier is implemented in the network hardware itself.

t	Event	CPU A Data Cache	CPU B Data Cache	Location X in Main Memory
				0
↓	CPU A reads X	0		0
	CPU B reads X	0	0	0
	CPU A writes 1 to X	1	**0**	1

Table 2.2: Cache contents for two processors demonstrating the need for cache coherence (reproduced from Figure 5.35 of Patterson and Hennessy [23]).

2.1.2.4 Cache Coherency

Caches are common in all types of modern computer. In a multi-core setting, this causes a problem because each processor could have a different view of the same memory location. This is even true of a shared memory UMA system like that of Figure 2.8 (assuming the processors here had caches). Consider the flow of events for a two-core system given in Table 2.2. Both CPUs read the same memory location, but the change in value performed by CPU A is not propagated to CPU B.

Maintaining *cache coherence* avoids the problem illustrated in Table 2.2 by ensuring that all caches have the same view of a particular location. One of the most popular cache coherence protocol is "snooping", where each cache maintains a record of the sharing state of a cache-line along in addition to the usual state [23, pg. 536]. A shared bus connects all of the caches together so that writes are propagated to all caches in the system that contain the same cache-line.

2.1.2.5 Inter-Process Communication (IPC)

A heavy emphasis has been placed on inter-processor communication in much of what has been described so far, but in real workloads the communication takes place between the *processes* and not the processors themselves. In a real system, inter-process communication will almost never carry a uniform cost. A processor with multiple cores will be able to used shared memory for IPC, but a system bus or a network protocol will be required for processes running on another physical chip. Consider the NUMA architecture presented earlier (Figure 2.9), here processes running on P_1 and P_2 will be able to communicate far more efficiently than processes on P_4 and P_5.

Communication is also a sequential task. Preparing the data for transmission (and subsequently decoding it on the receiving processor) may benefit from the use of SIMD

instructions, for example; however, the transmission itself contributes heavily to the s term in equation (1.6). A mutex- protected resource on a NUMA system could lead to a large s contribution and hence suffer a large performance degradation.

2.1.2.6 Load Imbalance

Improperly balanced workloads can lead to some processors being idle whilst others are busy completing their task [55, pgs. 135–136]. Recall the architecture of Figure 2.5. When it was first introduced, an example was given where a workload was uniformly distributed over all processors, but that P_1 and P_2 offered less performance than the rest. In this situation, the other processors would finish their work before P_1 and P_2 and thus be idle. The total idle time incurred contributes to an overall drop in processor utilisation.

SPMD architectures are particularly sensitive to workload imbalance if the individual computers that comprise the machine are of varying ages and technology levels. Regular sequence points can make this worse as it can skew the ratio of idle time to processing time for the worse on the faster cores.

Kumar, Grama, Gupta, *et al.* [55] also identify a special case that can cause significant performance degradation: parts of a program that cannot be parallelised *at all* might require that a single processor picks up the task, leading to *all other processors in the system* being idle simultaneously.

2.1.3 Communication

2.1.3.1 Chip Multi-core Systems

Processors in single-core systems have full access to the entire system memory-map. In addition to the system memory, this memory-map will include the registers of any devices the processor is expected to control. Some peripheral devices may include small sets of memory of their own (such as the transmit/receive buffers of a communications controller).

Moving data between these smaller memories and the system memory is an unnecessary burden for the system processor. Typically a Dynamic Memory Access (DMA) engine will be present to perform the copying of data instead of the processor. Once the copy has been completed, the DMA engine will raise an interrupt request on the processor so that the freshly copied data can be processed. This is often called *latency hiding* because the processor can pause the task that requires the data until the DMA issues the interrupt, and perform other useful work whilst the copy is taking place. From the perspective of the paused task, it appears as if the operation has taken zero time.

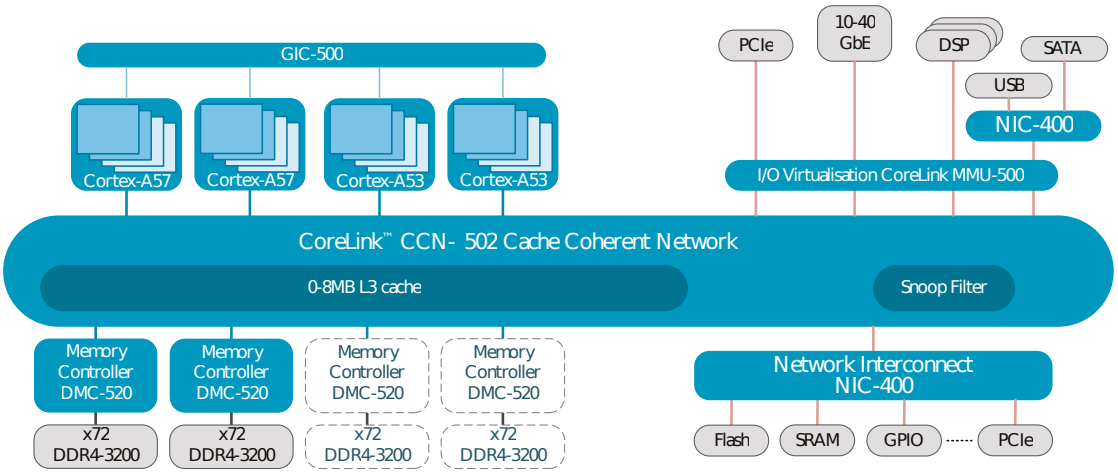


Figure 2.14: ARM CoreLink™ CCN-502 system diagram showing an asymmetric multi-core system (comprised of Cortex-A57 and Cortex-A53 processors) sharing a cache-coherent system bus (taken from ARM Ltd. [56]).

Chip multi-core processor (CMP) systems are essentially the same but with multiple processors sharing the bus. Figure 2.14 shows an asymmetric multi-core system with ARM Cortex-A57 and Cortex-A53 processors sharing the system bus. With many processors in the system, cache coherency (described earlier) becomes an important issue to mitigate. This system enforces cache coherency by embedding the L3 cache (fitted with a snooping filter) into the network itself. Memory accesses performed by all processors (including peripheral register accesses) are therefore guaranteed to be coherent between all other processors.

2.1.3.2 Networks

Large computer systems are built from multiple chips and, often, multiple smaller computers. In these cases, the "System Bus" of Figure 2.9 is replaced by a network. The bandwidth and latency of these networks varies depending on their intended purpose, however the overhead of the network will always degrade performance below that of a system bus. Chip-level networks are often preferred over dedicated wiring, despite the loss in performance, because they can simplify the design overall by providing a common interface to all blocks [57].

Inter-processor networks in the case of systems containing two physical processor chips, such as a desktop workstation or a headless server (i.e., without video output), will have a large bandwidth and a low latency. On the other hand, a cluster consisting of multiple discrete servers will be connected by a network that, while still high-performance, will be slower overall than the inter-processor network.

Networks are constructed from *endpoints*, *routers*, and *switches*. Endpoints are the components that the network is to connect together—the sources and sinks of data.

Routers and switches work closely to control how this data flows from the source endpoint to the sink. A router will inspect part of the packet for routing information (e.g., a destination address) and set the switches to control the flow of data. This close relationship means that it could be argued that switches are a sub-component of a router.

A *crossbar switch*, shown in Figure 2.15, is commonly used in both intra- and inter-chip networks. A crossbar switch is capable of connecting any of its n inputs to any one of its m outputs at a time [58]. For this reason, the shorthand $n \times m$ crossbar is often used.

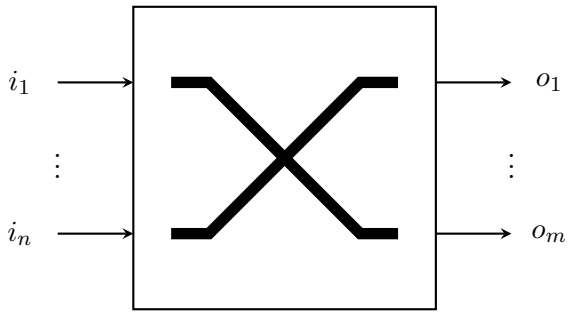


Figure 2.15: An $n \times m$ crossbar switch with n input ports and m output ports.

Network traffic consists of a flow of *packets* that are themselves a float of *flits*. A packet is a complete package of data that a source wishes delivered to the appropriate source. A flit is the smallest unit of data router can process, which is usually equal to its word-size. Consider that a source processor constructs a packet of N_p bits and issues it to network with an flit-size of N_f bits: each router will take N_p/N_f clock cycles to fully handle the packet.

Routers typically employ one of the following switching methods:

Circuit switching reserves the complete route from the source endpoint to the sink and allows a packet to flow through it uncontested, similar to a telephone exchange. Any other route requiring the same switches will be blocked until this *virtual channel* has been closed. For example, consider that a virtual channel is established between nodes P_1 and P_3 of Figure 2.16 (shown in blue), and then P_5 wishes to send a message to P_3 as well. The only route it can take is via P_6 (shown in green) because P_2 is reserved as part of the first virtual channel. The red route shows the failed first routing attempt.

Packet switching does not require that a virtual channel is established and instead allows the data to take whichever route is most appropriate at the time it is needed. For example, consider again the previous example transmission between P_1 and P_3, but in this case a large amount of data is being sent. In a circuit-switched system, the virtual channel would remain present for the entire time that data is being sent. However, in a packet-switching setting, the large transmission would

be broken up into network packets that each hold a portion of the data and would be routed individually. P_5's message could be interleaved with the flow between P_1 and P_3 through P_2 under this regime. This is a more complicated approach because it requires that the router in the sink endpoint correctly re-assembles the full message before delivering it to the processor.

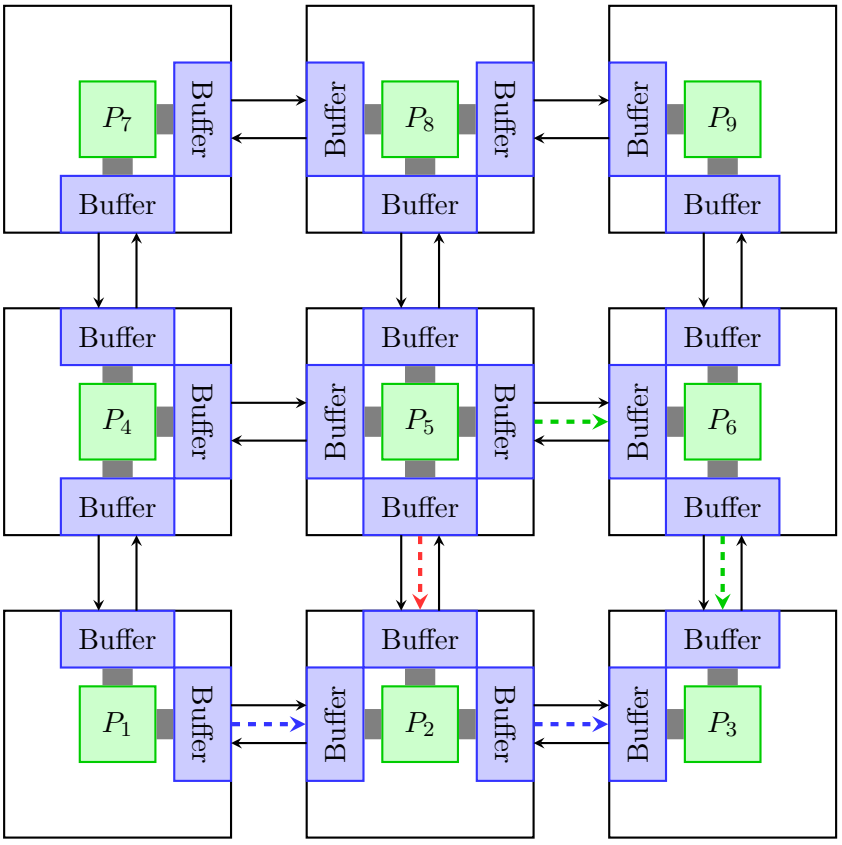


Figure 2.16: Example of a network of crossbar switches with buffered I/O ports connecting a set of processors.

In addition to these two switching strategies, there are also three commonly-used buffering (or flow control) strategies. Each tile in the example system of Figure 2.16 contains a processor, an $N \times N$ crossbar switch represented by the thick grey lines, and a buffer on each I/O port. The size of these buffers (which are seperate for the input and output ports) governs which strategy can be applied.

Cut-through flow control (also *wormhole routing*) [58] only requires that a single flit is buffered at each switching stage, effectively acting as a distributed pipeline for the message. The channel does not need to be reserved ahead of time, making it compatible with both circuit- and packet-switched networks. The key drawback with this approach is that multiple packets in transit can collide, at which point one must cease its progress to allow the other to complete. Once the collision has been resolved, the second packet can continue its journey. This drawback is similar to that of circuit switching.

Virtual cut-through resolves this issue by increasing the buffer size to hold a complete packet. When there is no contention on the network, it behaves identically to cut-through routing by only buffering a few flits at a time (i.e., forming a pipeline). However, if a packet encounters congestion on the network, the routers have sufficiently large buffers such that the last one in the chain can receive the packet entirely, thereby removing it from the network and preventing the congestion that would otherwise be present in cut-through routing.

Finally, *store and forward* is the simplest case for routing if the buffers are large enough to store a complete packet. A route will receive a packet in its entirety and then forward it onto the next router once the routing information has been analysed. Essentially, this is the same as a virtual cut-through system with large amounts of congestion.

2.1.3.3 Intra-chip Networks

Intra-chip networks tend to be the fastest of all networks and essentially serve as an alternative to a system bus [57]. ARM's CoreLink network shown in Figure 2.14 provides an example of the typical features of an on-chip network. Processors are connected to various on-chip peripherals and to the system main memory. From the perspective of each processor, this network will appear as a linear memory-map, similar to a standard system bus. Due to their close proximity to processors and memories, intra-chip networks usually either support or provide cache-coherency.

These networks commonly use crossbar switches and virtual cut-through routing to maximise bandwidth and minimise latency [15, 17, 18]. However, alternative approaches are also viable. The SPARC M7, for example, uses a custom 'multi-stage data network' which consists of separate request and response stages [19]. The request network uses a 4-ring topology, while the response network is point-to-point. The combination affords this on-chip network a bandwidth of about 4Tbps.

Another multi-stage design is the *swizzle-switch network* [59] which divides network access into *arbitration*, *data*, and *release* stages, allowing it to support various quality-of-service (QoS) levels. Higher QoS traffic will always win the arbitration stage over lower levels, but a second arbitration stage uses least-recently-granted to ensure fairness in each QoS level. Essentially, this network functions as an advanced crossbar capable of providing a bandwidth of up to 4.5Tbps.

2.1.3.4 Inter-chip Networks

Networks for connecting multiple chips in a single machine (e.g., multiple processors fitted to a single server motherboard) are usually point-to-point, with crossbar-switch-backed routing stages in each endpoint. Networks at this level will usually serve as the

off-chip peripheral bus for the processor and are therefore not guaranteed to provide cache-coherency. Additionally, despite contributing to the system memory-map, it is highly unlikely that they will offer a path to main memory.

Intel QuickPath Interconnect (QPI) is used to create a network of processors capable of sharing memories and devices to which they are connected in modern desktop and server computers [60]. Figure 2.17 provides a typical use-case for QPI. All processors are allowed to access the memories and chipsets attached to any other processor, thus creating a NUMA architecture. Each endpoint of the network contains a cross-bar switch to support this.

QPI is specifically designed for multi-chip systems, hence cache-coherency is built into the protocol stack. This applies to both memories and peripherals hosted by any of the chipsets. That is, P_1 is permitted the use of a peripheral hosted by the chipset connected to P_3 in addition to the local memory of P_3, with coherency mediated by the QPI hardware itself.

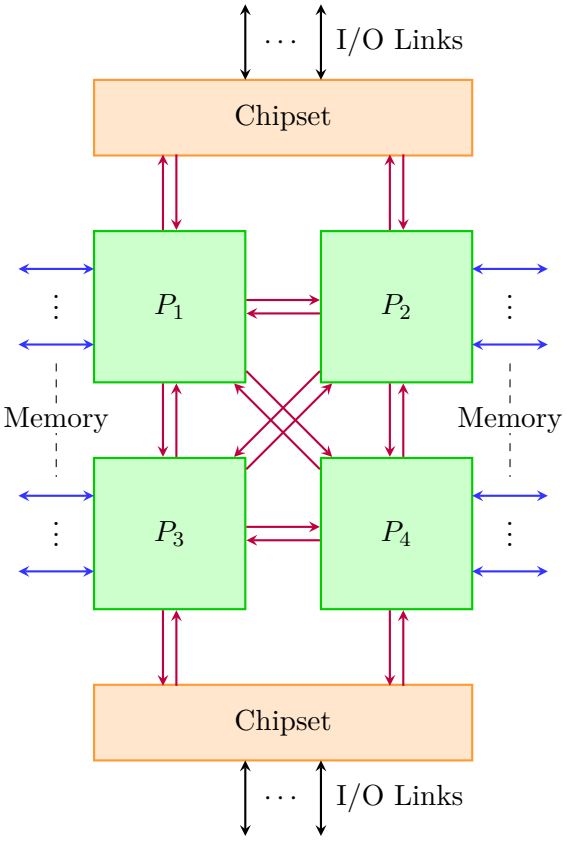


Figure 2.17: A typical QPI use-case produced from Figure 6 of Intel Corporation [60]. Memory connections are shown in blue and QPI connections are shown in purple.

HyperTransport serves a similar purpose as QPI for AMD processors, but is more general as it has been used in Cisco routers, and IBM and Sun Microsystems severs as well [61]. A similar multi-processor use-case for HyperTransport is show in Figure 2.18. This network

also uses point-to-point connections but operates slightly slower at 22.4GB/s (in contrast to 25.6GB/s for QPI) and does not provide hardware cache-coherency support. As can be seen from Figure 2.18, HyperTransport can serve as a high-speed system network hosting multiple bridges into other communications protocols. Whilst this is possible with QPI, peripheral management and communications protocol bridging is delegated to the chipset.

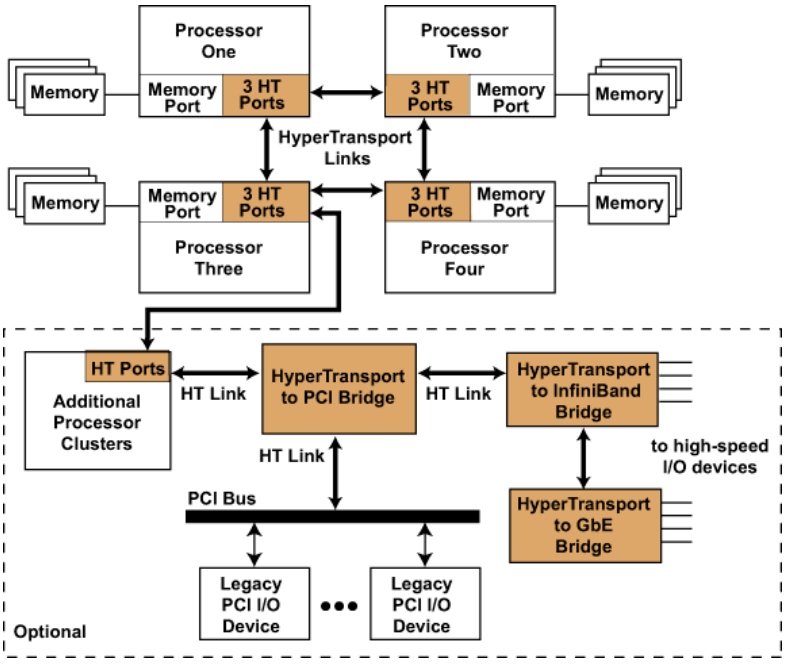


Figure 2.18: An inter-processor use-case for HyperTransport showing how processors connect to each other and to system peripherals (Figure 11 of [61]).

2.1.3.5 Cluster Computer Backbone Networks

Most distant from individual processor cores are the inter-computer networks which serve as *backbones*. These networks are present in a wide range of computing systems, from supercomputers and cluster computers to office networks. They incur greater latency and offer low bandwidth than the networks previously introduced, and for this reason provide no guarantee of cache-coherency.

Backbone networks usually offer a great degree of flexibility in their connection topology due to their scale. Careful selection of a topology is important to minimise the number of hops (i.e., routing stages) traffic must take to traverse the network [58].

Blue Gene/Q is permeated by a 5D torus network of processing and I/O nodes (for storage) [62]. Each link on this network has a transfer rate of 2GB/s and are controlled by *messaging units* which are integrated into the same die as the processors. Despite this high-locality, there is no attempt to maintain cache-coherency globally across the machine.

Myrinet [63] has been specifically designed with cluster computing in mind and focuses on reducing the cost of communication at each stage, aiming to reduce the s contribution of equation (1.6) as much as possible. A large focus has been placed on reducing the number of buffers through which processes wishing to transmit and receive data must copy it. On platforms that support it, Myrinet drivers are capable of copying data from user-space memory directly into the transmit buffer on the network-interface-card (NIC). Additionally, Myrinet traffic is *source-routed* which means the complete path that a packet must take is determined at the source. Each receiving node strips the top word from the header and uses this to route the packet to the next node. Once all routing information has been removed, the packet has arrived at its destination. This heavy focus on minimising serial overhead and simplistic routing has made Myrinet viable as a host for peripheral devices [64].

Many cluster computers (and networks of computers in general) use Ethernet rather than a special-purpose network. It was originally designed as a general-purpose network, and this ubiquity has driven efforts to improve the performance. Recent advancements have increased the effective bandwidth close to that of Myrinet, making it a suitable (and often cheap) alternative in cluster computers [65, 66]. Despite these advancements, Ethernet still has a comparatively high latency when compared to Myrinet, however continual efforts are reducing this difference further [67].

2.1.4 Software and Frameworks

2.1.4.1 Asynchronous I/O

Asynchronous I/O is a software system that implements latency hiding described earlier. It is usually implemented using an *event loop* and hence does not provide a parallel programming interface to the programmer. Instead, parts of the program will request that an I/O operation is started, leaving the framework to issue an event once it has completed. Often this will be implemented by the framework spawning one or more threads to handle the I/O operations in a blocking manner, reporting the results to the main thread once complete. This is an implementation detail that is hidden from the main program.

Node.js [68] is an implementation of the JavaScript language commonly used to drive the back-end of modern websites, and is heavily rooted in this approach. The event loop is spawned automatically when the interpreter starts, allowing Node.js programs to schedule and respond to events immediately. Listing 2.1 shows an example of how one might read a file using the FileSystem API [69] that forms part of the standard library of Node.js. Line 4 contains the call to `fs.open` which accepts an *anonymous function* (i.e., a function defined in-place) that will be called when the operation has completed. `fs.open` then returns immediately allowing the program flow to continue. In this case,

there is no more code to execute, so Node.js idles until there is an event to dispatch. Once the file has been opened, the anonymous function is called with the handle of the file that has been opened.

```
1  var fs = require("fs");
2  var buf = new Buffer(1024);
3
4  fs.open('input.txt', 'r', function (err, fd) {
5     if (err) {
6         return console.error(err);
7     }
8
9     fs.read(fd, buf, 0, buf.length, 0, function (err, bytes) {
10       if (err) {
11          return console.log(err);
12       }
13
14       console.log(bytes + " bytes read");
15    });
16 });
```

Listing 2.1: Reading a file using the asynchronous API provided by Node.js.

Python 3.4 offers a similar capability in the `asyncio` module of the standard library [70]. Instead of invoking callbacks as with the Node.js, `asyncio` switches between various *co-routines* assigned to each I/O resource. When a co-routine would perform a blocking operation, such as reading from a file, it instead implicitly de-schedules itself. The main event-loop flags this co-routine as blocking on an I/O stream and schedules another co-routine that is not waiting on any resources.

It is important to note that in both of these APIs, there is only a single thread of execution in the program. If an anonymous function (in the case of Node.js) or a co-routine (in the case of Python) performs another form of blocking operation, such as iterating over a very large number of items, the handling of any I/O tasks that complete in the meantime will be equally delayed.

2.1.4.2 Multiple Threads

A process will always have at least one thread associated with it. On small embedded systems built around microcontrollers, this is explicit because the process has exclusive access to the hardware, which is only capable of hosting a single thread. On larger systems capable of hosting a modern operating system, this fact is usually not obvious to the programmer.

Spawning multiple threads inside a processes is typically only achievable on larger operating systems, but is not limited to systems with multiple process cores. Real-time

Operating Systems (RTOS') will provide some level of thread support to aid with latency hiding from slow I/O resources and to simplify authoring of the software by providing some encapsulation. The same is true of desktop and server operating systems running on systems with only a single core, but this is becoming exceedingly uncommon.

Regardless of the architecture of the host system, threads allow various parallel programming techniques to be used, from latency hiding in the simplest case, to true parallel speed-ups in the most complex. Synchronisation, as mentioned earlier, plays an important role here to eliminate race conditions on shared resources. Consider a thread that has been spawned to handle blocking I/O operations to hide their latency. There must exist a mechanism for it to communicate with the main thread, else they may access shared resources simultaneously and cause an irrecoverable fault.

`pthreads` is the POSIX threads library forming part of the Linux operating system [71]. It is written in 'C' and has been also been ported to other operating systems, making it a widely available threading library. For synchronisation between threads, it relies primarily on mutex locks and related variants.

```
#include <vector>
#include <thread>
#include <future>
#include <numeric>
#include <iostream>

void accumulate(std::vector<int>::iterator first,
                std::vector<int>::iterator last,
                std::promise<int> accumulate_promise)
{
    int sum = std::accumulate(first, last, 0);
    accumulate_promise.set_value(sum);  // Notify future
}

int main()
{
    std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };

    std::promise<int> accumulate_promise;
    std::future<int> accumulate_future =
        accumulate_promise.get_future();
    std::thread work_thread(
        accumulate, numbers.begin(), numbers.end(),
        std::move(accumulate_promise));
    //...do some other useful work...
    std::cout << "result=" << accumulate_future.get() << '\n';
    work_thread.join();  // wait for thread completion
}
```

Listing 2.2: Computing the sum of a set of numbers using a C++ worker thread.

A more recent cross-platform multi-threading tool is `std::thread` in the C++11 standard library [72]. Mutex locks are also available with this approach. However, a promise/future interface is also provided which can make latency-hiding and synchronisation with long-running computations simpler. A `std::promise` [73] encapsulates an agreement (of sorts) that a value will be available for, or from, a thread at some point. A `std::future` [74] provides the read access component of a promise.

Listing 2.2 provides an example of a promise/future relationship being used. Lines 17–21 allocate the promise that will wrap the result, the future that will be used to retrieve it, and then create the thread that will perform the work. The `accumulate` function (lines 7–11) then performs the sum and sets the result in the promise. Line 25 prints the result of this sum using the future obtained on line 21 and prints it to the terminal. It is important to note that in this example, `accumulate_future.get()` blocks until the worker thread has completed its task. In a real-world situation, additional useful work would be performed between starting the thread and waiting for the result.

```
1  #include <vector>
2  #include <thread>
3  #include <future>
4  #include <numeric>
5  #include <iostream>
6
7  int accumulate(std::vector<int>::iterator first,
8                 std::vector<int>::iterator last)
9  {
10     return std::accumulate(first, last, 0);
11 }
12
13 int main()
14 {
15     std::vector<int> numbers = { 1, 2, 3, 4, 5, 6 };
16
17     std::packaged_task<int(std::vector<int>::iterator,
18                            std::vector<int>::iterator)>
19         accumulator(accumulate);
20     std::future<int> accumulator_future =
21         accumulator.get_future();
22     std::thread work_thread(
23         std::move(accumulator), numbers.begin(), numbers.end());
24     //...do some other useful work...
25     std::cout << "result=" << accumulator_future.get() << '\n';
26     work_thread.join();
27 }
```

Listing 2.3: Computing the sum of a set of numbers using a C++ packaged task.

Promises and futures are a useful technique, but as Listing 2.2 illustrates, code must be explicitly written to support them. However, C++11 introduces another class, `std::packaged_task` [75], which can wrap a function and provide a promise/future relationship automatically. Listing 2.3 shows the same program as Listing 2.2 but reformulated to used a `std::packaged_task`. The most important distinction between the two is that the `accumulate` function (lines 7–11) does not need to accept the promise as a parameter. This allows worker functions like `accumulate` to be re-used more often in the program, perhaps in a synchronous setting for small data-sets and in a separate thread when large.

A similar feature is integrated into the C# language. The `async` and `await` keywords [76] provide an interface similar the `std::packaged_task` but without the need to explicitly allocate a thread. `async` methods return a promise that can be interrogated in a non-blocking manner to allow useful work to be completed until the result is ready. The `await` keyword then functions equivalently to the `std::future::get` method, by blocking until the result is ready.

The threading approaches mentioned up to this point have focused on latency-hiding and coarse-grained parallelism. OpenMP provides an alternative approach by introducing commands into the compiler to achieve finer-grain parallelism [77]. Listing 2.4 shows an example (based on Figure 3.10 from Chapman, Jost, and Pas [77]) where a `for`-loop that implements a matrix-vector product is explicitly marked for unrolling. The `#pragma` command on line 6 requests that the compiler generates multi-threaded code, and identifies which variables will be shared across all threads (i.e., require synchronisation) and which are thread-local. This approach is less general than the previously mentioned methods, but can yield significant performance increases in structured computations.

```
1  void mxv(int m, int n, double * restrict a,
2           double * restrict b, double * restrict c)
3  {
4      int i, j;
5
6  #pragma omp parallel for default(none) \
7          shared(m, n, a, b, c) private(i, j)
8      for (i = 0; i < m; i++) {
9          a[i] = 0.0;
10
11         for (j = 0; j < n; j++)
12             a[i] += b[i * n + j] * c[j];
13     }
14 }
```

Listing 2.4: Using OpenMP to unroll a sequential loop into multiple threads.

2.1.4.3 Multiple Processes

Multi-process applications provide similar performance benefits to those described in the previous section, with the added benefit that the computation can be distributed across many physical computers using an SPMD or MPMD architecture. Individually, the processes may be multi-threaded to boost the performance of the part of the problem. Multi-process applications can also be executed on a single machine where they function similarly to a multi-threaded application. Typically, the overhead of multi-process computation is more significant than spawning multiple threads, but provides the potential for increased fault tolerance; if a single process crashes or halts, the overall application can (potentially) continue to function.

The Message-Passing Interface (MPI) is a multi-process library that is commonly used for scientific computation [78]. It has been standardised and has implementations available for many operating systems. The original use-case was similar to that shown in Listing 2.4, with the two-sided send/receive communications of MPI-1 taking the place of the shared variables in the `#pragma` directive. Since then, MPI-2 has added parallel file I/O routines and one-sided put/get communications, which allows MPI applications to take on an MPMD structure when more appropriate. Dynamic process management in MPI-3 takes this further still, allowing ad-hoc MPMD structures.

Part of the reason for the wide-scale adoption of MPI is the process manager, `mpich`, which allows software binaries to be distributed over all of the individual computers forming a cluster machine. Submitting a job to the cluster machine therefore reduces to an invocation of `mpich` with the appropriate binary and arguments. `mpich` then takes care of synchronously starting the programs on the various machines.

An alternative to MPI that is more common in production systems and less common in simulation systems is ZeroMQ [79]. Unlike MPI, ZeroMQ does not aim to provide a complete system capable of supporting multi-process applications. Instead, it is a library that provides a set of communication topologies that can be used together throughout an application. One-to-one, one-to-many, many-to-one, and many-to-many topologies are provided, allowing applications to have heterogeneous communications throughout, using the most appropriate for each set of components.

ZeroMQ does not provide a process manage equivalent to MPI and relies on the system architects to design a solution to start the processes. A key advantage over MPI is that ZeroMQ allows for code to start small using shared memory communications in a single process, but permits it to grow into special-purpose multi-process applications optionally spread over multiple machines (using TCP-backed communications) without requiring significant changes to the code.

2.2 Neuromorphic Computing

2.2.1 Biology

Neuromorphic systems are inspired by the resilience, power-efficiency, and massive parallelism of mammalian central nervous systems. Using the human brain as an example: it contains an extraordinary number of neurons (about 10^{11}) embedded in a highly connected network (about 10^{15} synapses), yet only consumes around 25W and is composed from elements that operate relatively slowly (around 10Hz) [35].

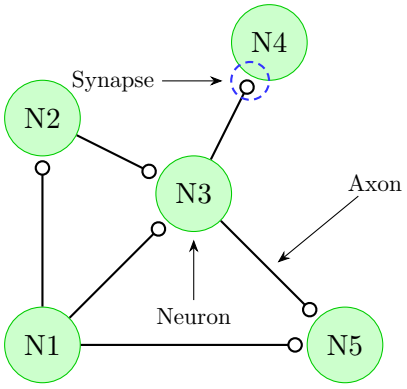


Figure 2.19: A small neural network.

Figure 2.19 shows an annotated visual representation of a small neural network. *Neurons* are the fundamental processing elements of the network. They can accept many thousands of inputs from other neurons, and can connect to many thousands of other neurons. *Synapses* serve as the inputs to a neuron and can be either inhibitory or excitatory, which can roughly be considered as 'active-low' and 'active-high' in digital electronics. *Axons* are the wires through which the neural signals flow, and behave similarly to transmission lines.

There are a range of methods through which neurons communicate, but only *spiking* neural networks are considered here. In these systems, neurons *fire* when their particular conditions are met, and emit very small voltage spikes through their axons. These spikes eventually reach the synapses, and cause an emission of *neurotransmitter*, which is a chemical process that excites (or inhibits) the target neuron. Neurotransmitter is a finite resource that must be recouped after use. This delay is called the *refractory period* and places a natural limit on the number of effective spikes that can be discretely identified by the receiving neuron. Typically, a single spike causes only a small release of neurotransmitter, thus neurons tend to produce chains of several spikes.

Neurons are usually modelled by differential equations that relate their inputs to their outputs. Two commonly used models are the leaky integrate-and-fire (LIF) [80] and Izhikevich [81] models. Both compute the weighted sum of all incident spikes to determine the *membrane potential*, which is a voltage that decays over time; excitatory

synapses increase this potential whereas inhibitory synapses decrease it. If the membrane potential exceeds a specific threshold, the neuron fires a spike of its own, which costs energy and hence decreases the membrane potential back to below the threshold. The Izhikevich model differs from the LIF model here by introducing a "recovery term" into the differential equation to model the refractory period. Over time, the membrane potential naturally decays in both models without a sufficient volume of spikes.

Adjusting the weights applied to the sum of incident spikes allows neurons (both modelled and biological) to alter their behaviour over time. This learning process is called *plasticity*. Spike-timing-dependent plasticity (STDP) is a method commonly used to achieve this [82]. The synaptic weights mentioned above are tuned according to the difference in firing rates between the *pre-synaptic* neuron (i.e., the source) and the *post-synaptic* neuron (i.e., the neuron receiving the spike). Weights are increased if the spikes arrive before the post-synaptic neuron fires, and decrease if they arrive after [83].

2.2.2 Hardware

2.2.2.1 BrainScaleS

Of the hardware platforms mentioned in this section, the FACETS system developed for the BrainScaleS project [84] is the farthest from any parallel hardware previously described. The core architecture consists of a full *wafer-scale* system where each neuron is a physical transistor circuit [85]. For this reason, the architecture is very power-efficient at the cost of being inflexible.

The system employs a multi-tier communication system, shown in Figure 2.20, to connect the neuron models together both locally and between wafers. Top-level communication between wafers and host computers are mediated by FPGAs. An application-specific 'digital network chip' (DNC) connects multiple high input-count analogue neural network (HICANN) modules together and to the FPGA network. The traffic flowing generated by the HICANN modules uses address-event representation (AER) to represent neural spikes [86]. AER will be explained in Chapter 3.

This system has been designed to simulate neuron models 10^4 times faster than biological real time. Additionally, the wafer-scale approach allows a very high connection-density, as each wafer holds 180k physical neuron models embedded in 40M synapses [86].

2.2.2.2 TrueNorth

IBM's TrueNorth platform uses a highly-specialised *core*, shown in Figure 2.21a, which hosts a set of 256 neuron units (i.e., outputs) connected to 256 axon units (i.e., inputs) via a 'synaptic crossbar." Each neuron unit is configurable and hosts a "a wide variety of

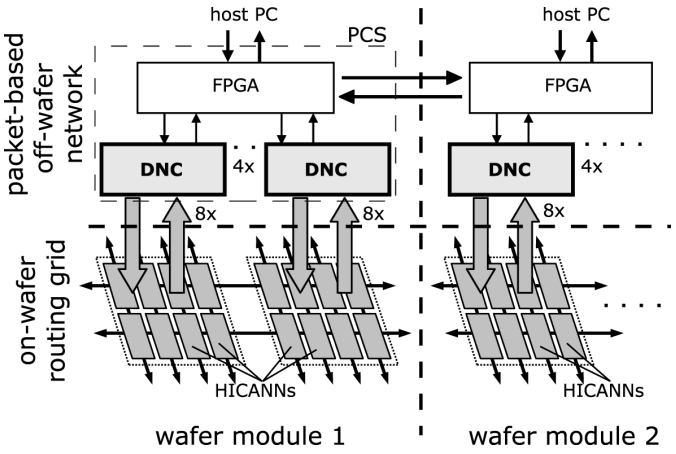
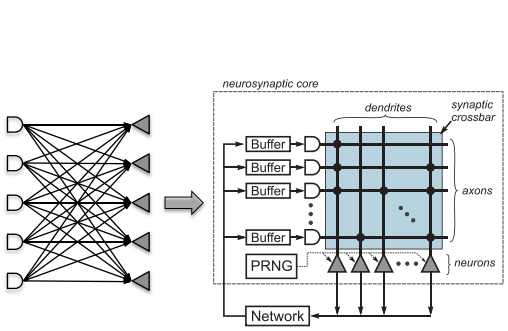
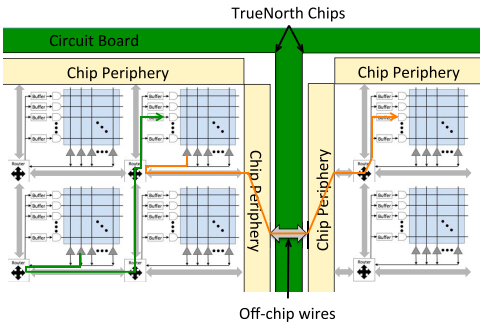


Figure 2.20: Logical structure of the BrainScaleS off-wafer hierarchical communication network (taken from [86]).

computational functions and biologically relevant spiking behaviours" [87]. The neuron models are updated using a 1kHz clock which permeates the entire fabric to enforce global synchronisation at this rate. Between the clock pulses, each core is allowed to operate in parallel, executing event-driven behaviours.



(a) TrueNorth core showing the axon/neuron connection fabric.



(b) Network of TrueNorth cores both inside a chip and at the boundary.

Figure 2.21: Structural diagrams of the TrueNorth platform (taken from Akopyan, Sawada, Cassidy, *et al.* [88]).

Each TrueNorth chip contains a 64×64 matrix of the cores shown in Figure 2.21a, resulting in one million neurons and 256 million non-plastic synapses per chip [88]. The core matrix can cross chip-boundaries as shown in Figure 2.21b allowing larger neural networks to be constructed from multiple chips. TrueNorth uses spiking neural models, as with the BrainScaleS system. To date, the largest TrueNorth system constructed comprised 16 chips for a total of 16 million neurons.

2.2.2.3 BlueHive

BlueHive is constructed from readily available Terasic DE4 development boards, built around an Altera Stratix IV 230 FPGA [89]. The architecture shown in Figure 2.22

implements the Izhikevich model with a 1ms integration time-step used to solve the differential equations. Each FPGA is capable of hosting 64k neurons with 64M synapses, leading to a fan-in of around 1k synapses per neuron. Several boards can be connected by a 6GB/s network, enabling the simulation of larger neural networks. The largest BlueHive system to date consists of 4 FPGAs, thereby supporting 256k neurons and 256M synapses. Synapse data is stored in an FPGA-local SDRAM affording the system a level of run-time configurability despite being constructed from FPGAs.

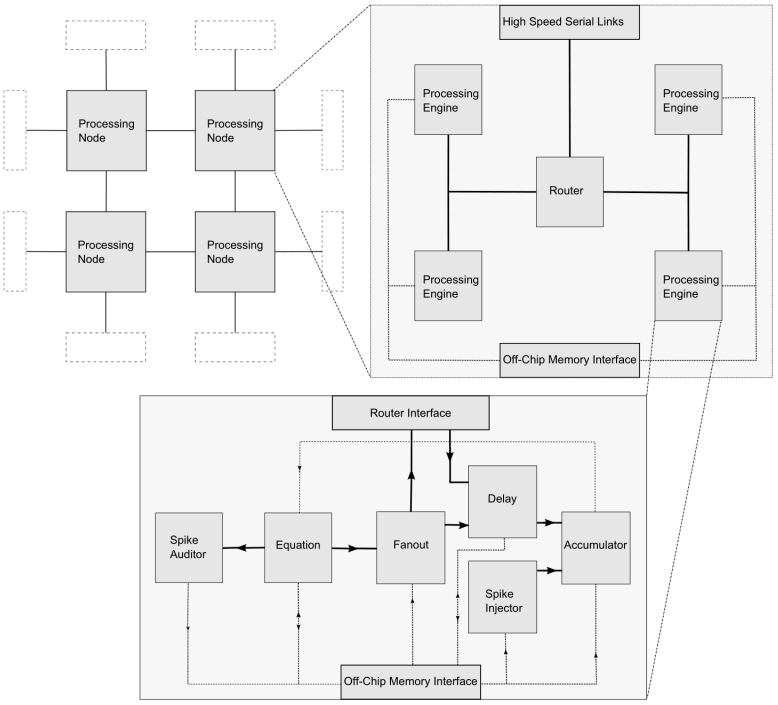


Figure 2.22: BlueHive system-level diagram (taken from Moore, Fox, Marsh, *et al.* [89]).

The original design of BlueHive delegated the neural processing to an Altera NIOS soft-processor before replacing this with a special-purpose processing pipeline. All FPGA boards in the BlueHive system are connected to a secondary FPGA board and a Mini-ITX computer which work together to program the 'worker' FPGAs in parallel. This allows BlueHive to use a neuron model other than Izhikevich, or even to replace the neuron pipeline with a NIOS processor as before. Of the three architectures discussed so far, BlueHive is the only system capable of executing simulations with non-neural behaviour.

2.2.2.4 SpiNNaker

SpiNNaker is a custom ASIC designed to support neural network simulation using readily-available ARM968 processors [35]. A full sized SpiNNaker machine will contain over a million processors distributed over approximately 57k chips, or *nodes*. A

block-diagram of a SpiNNaker node is given in Figure 2.23. Each processor is capable of hosting 10^3 neuron models with a fan-in of, nominally, 10^3 synapses each. Overall, a SpiNNaker machine can support 10^9 neurons with 10^{12} synapses operating in biological real-time.

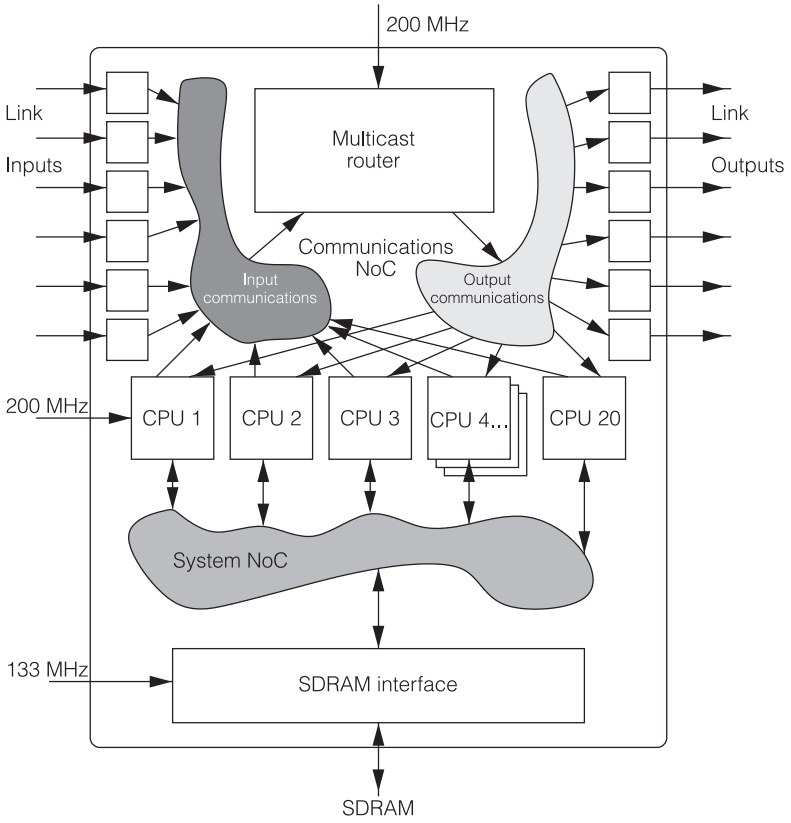


Figure 2.23: SpiNNaker chip block diagram (taken from Plana, Furber, Temple, *et al.* [90]).

Each node has a local 128MB SDRAM wire-bonded in the same package as the node die and uses a self-timed network to communicate to its six immediate neighbours [91]. The self-timed nature of this network means that the communication rate naturally falls as the processors apply back-pressure, helping to self-regulate the flow of traffic through the system. The SpiNNaker nodes are embedded in a 2D torus network leading to a bisection bandwidth of 4.8Gpackets/s [6]. SpiNNaker serves as the underpinning technology for the thesis, and is explained in much more detail in Chapter 3.

2.2.3 Software

Neural simulations are the flagship applications for neuromorphic software. Each simulator provides a range of models for neurons and neuron-synapse interactions, and aims for a specific level of biological detail. As with the previous section, here we do not aim to give a detailed description of neural simulation, but provide some insight into

how current neural simulations enhance their performance using the parallel computing techniques described earlier.

The selection of simulators described here is widely used in computational neuroscience. Where appropriate, performance figures are given, but as each simulator targets a different level of biological accuracy and scale, a like-for-like comparison is not necessarily achievable.

2.2.3.1 NEURON

NEURON [92] is a well-established simulator designed for a high level of biological fidelity. Simulations may range from parts of single neurons up to circuits containing a small number of cells. A particular emphasis has been placed on the dynamics of nerves, where very detailed models based on electrical transmission lines have been derived [93]. Neuron models consist of multiple *compartments* into which cell membrane potentials are bundled. An extra node with zero area is added to the transmission line model to serve as the endpoint incident with the target neuron (i.e., the synaptic connection itself). This approach slightly degrades the overall computation efficiency, but is justified by the biological detail it provides.

NEURON itself is written in 'C' and ships with several pre-defined models for neurons and synapses which treat cell membrane potentials as sets of simultaneous differential equations. Additional models can be defined using the NMODL language, which is somewhat 'C'-like [94]. An interpreter in NEURON translates the NMODL code into a 'C' program which is linked against appropriate NEURON libraries.

Whilst parallel execution is not a key design intent for NEURON, parallel simulations with it have been achieved using a Blue Gene/L. The Blue Brain project [95] ran NEURON with automated fitting algorithms that inserted ion channels[3] into existing neuron models, and then tunes the various parameters accordingly. On standard computing hardware, fitting a single neuron with this improved functionality can take up most of a day. Each neuron in this case is 'morphologically and electrically unique' [95] meaning that fitting parameters for one neuron cannot be used or another. However, the parallelism provided by the Blue Gene/L allowed around 10k unique neurons to be fitted in just one day.

2.2.3.2 NEST

The NEural Simulation Tool (NEST) focuses on the size and structure of spiking neural networks rather than offering precise biological detail for individual neurons [96]. Networks are constructed from *nodes* which are the fundamental units in NEST. Nodes may

[3] A physical part of a synapse that controls the flow of neurotransmitters into a neuron.

be a neuron model or a sub-network, and communicate with other nodes by transmitting specific types of *event* through inter-node *connection* objects which model synapse behaviours. In the simplest case, this model is simply a weight and a delay, but spike-timing dependent plasticity (STDP) is also available.

Networks in NEST are not limited to a single neuron or synapse model thus enabling heterogeneous simulations [97]. Additionally, events may take one of two types: *discrete* for spiking neuron traffic, or *analogue* for AC/DC currents or membrane potentials.

NEST does not include a global event queue, instead constructing a full simulation from a set of *virtual processes* which each have control of a number of network nodes and connections. Both pthreads and MPI are used to manage these virtual processors, thus enabling NEST to trivially expand across a cluster machine [98]. All virtual processes perform neuron updates in parallel and then wait for a global synchronisation step (similar to a barrier). After this step, new events are allowed to be exchanged between the virtual processes. Due to the high number of events being passed around, a minimal event description is transmitted that requires reconstruction from local data in the receiver.

2.2.3.3 Brian

Brian [99] takes a slightly different approach from NEURON and NEST, by focusing on being simple to use so that time taken to develop new models is minimal [100]. Brian is written entirely in Python which has become popular for computational neuroscience (and scientific work in general). Any Python shell environment provides the user with the means to explore neural networks in an interactive manner. Neuron descriptions have therefore been designed to be as concise as possible. Despite their brevity, they are able to support short-term plasticity and STDP.

Neuron models are described using sets of differential equations which are written as string-literals in Python code. Brian parses these strings to construct a mathematical model of the desired behaviour which is then executed using the discrete event engine. Internally, the update computation makes use of vector operations to offset any performance impact that may be incurred by using Python. At present, there is no support for distributed simulations, but the developers are planning to add GPGPU support because graphics cards are more readily available [100].

2.2.3.4 PCSIM

Parallel Circuit SIMulator (PCSIM) follows design criteria that are a mixture of the three tools previously mentioned [101]. Similarly to NEST, PCSIM aims to allow exploration of large neural networks with models of less complexity than NEURON. As with Brian,

the Python front-end of PCSIM allows circuits to be explored interactively using a Python shell.

The PCSIM core is written in C++ for performance reasons and includes support for multi-threaded execution as well as distributed parallelism with MPI [102]. This has been demonstrated to support around 10^5 neurons, each with 10^3 synapses, on a cluster of 20 machines.

A set of neuron models is included in the C++ core, and users are able to add new models using either C++ or Python. Networks may be heterogeneous and may include 'abstract processing elements' which can accept discrete spiking signals, or analogue firing rates and membrane voltages. The abstract processing elements are a particularly powerful feature of PCSIM because the behaviour of a population of neurons can be delegated to another simulator (e.g., Brian) [102].

2.2.4 Languages

Neural description languages exist to simplify the specification of neural circuits and to aid collaboration by providing standardised file formats that can be shared via services such as Open Source Brain [103]. The landscape is rapidly changing but two key approaches have emerged: procedural formats that resemble code and declarative formats based on XML.

These description formats are not bound to any particular back-end or simulator, allowing models to be ported between them. These descriptions can be made graphically using tools such as neuroConstruct [104] that, whilst probably too cumbersome for experienced users, allows beginners to begin simulating networks quickly.

2.2.4.1 Procedural

PyNN [105] is the dominant procedural description language. It can target NEURON, NEST, PCSIM, and Brian but has also been extended to support the neuromorphic hardware platforms BrainScaleS [106] and SpiNNaker [107].

Whilst PyNN is often considered a language in its own right, it is in fact a Python library that provides the same interface regardless of the chosen back-end simulator [108]. It has been designed to provide a high level of abstraction which allows users to define (a) *populations* of neurons with statistical connection characteristics rather than requiring individual connections to be considered, and (b) *projections* which apply a similar concept to connections between populations.

As PyNN can target a range of simulators, consistency between them is extremely important. This is achieved by including a set of models in PyNN that perform identically,

whilst also providing an object-oriented means of defining custom models that will behave consistently between simulators. Internally, PyNN uses the most appropriate interface for the selected simulator to prevent any potential performance deterioration that may occur. To ensure the repeatability of experiments which may require the presence of random noise, PyNN includes a set of random number generators with means to explicitly, and easily, define a fixed seed. Where a simulator interface is not available for direct control, PyNN can generate NeuroML—a declarative format described in the next section.

2.2.4.2 Declarative

NeuroML is an XML-based language capable of describing neural systems at various levels of abstraction [109]. Level 1 includes MorphML which is used to describe the structure of individual cells (i.e., their morphology). Level 2 brings ChannelML which enables the description of synapse properties. Finally, Level 3 supports connections between populations, external electrical inputs, and geometric positions of components using NetworkML. Each level includes its own capabilities and also that of all levels below it, so a Level 3 NeuroML file is capable of describing a complete network of individually-defined cells. Descriptions are modular, allowing large descriptions to re-use smaller components where possible.

As with PyNN, NeuroML is not bound to a particular simulator and can therefore be used as a collaboration format. Unlike PyNN, however, NeuroML does not aim to wrap any simulators directly, but certain tools (like NEURON) can load NeuroML descriptions natively. The format is not completely general and only supports features that are widely available to maximise the number of simulators it can be used with.

NineML [110] emerged to address limitations in both PyNN and NeuroML. First, neither of these languages allow models to be expressed mathematically (i.e., like the input stage of Brian). Second, NeuroML does not include any connection algorithms that can generate complex layouts. Whilst PyNN does not explicitly support this either, it can be achieved using standard Python code because PyNN is simply a library. The key goal of NineML is to ultimately provide "a harmonized standard for representing most type of models, including large-scale networks" [111].

Low-Entropy Model Specification (LEMS) [112] provides a language to describe the low-level dynamics of biological system components, with a level of detail similar to Brian. LEMS defines both an XML standard for declarative descriptions and a reference implementation, PyLEMS, which supports an equivalent procedural description (similar to PyNN) and also includes a small simulator that can be used to verify the behaviour of individual models [113].

The most recent addition to the set of declarative languages is NeuroML 2, which aims to unify the capabilities of all these languages [114]. NeuroML is being used to form the basis, with a LEMS-compatible layer being added to support the description of individual devices. Overall, NeuroML 2 aims to increase the number of neural systems that it can describe to take a step towards being a single standard. Similarly, the device-level description syntax of NineML is compatible with LEMS to provide maximal collaborative potential [110].

2.3 SpiNNaker in Context

Table 2.3 provides a very brief comparison of the hardware architectures introduced in section 2.2.2. The table clearly shows that SpiNNaker offers the most scalability. It offers massive-scale parallelism in general-purpose cores, which makes it a viable platform for exploring how massively-parallel software can be structured.

Platform	**Flexibility**	**Scale**	**Simulation Rate**
BrainScaleS	ASIC—Low	180k neurons, 40M synapses	$10^4 \times$ real-time
BlueHive	FPGA—High	256k neurons, 256M synapses	real-time
SpiNNaker	CPU—High	10^9 neurons, 10^{12} synapses	real-time
TrueNorth	ASIC—Low	16M neurons, 4bn synapses	real-time

Table 2.3: Comparison of selected neuromorphic hardware platforms.

Data is distributed with very high-granularity across SpiNNaker, which means that traditional parallel synchronisation techniques are not necessarily required. All resources that are shared between multiple processors are already hardware-arbitrated in the chip which inherently resolves any potential race conditions.

For SpiNNaker to be able to support non-neural computation, problems must be slightly reformulated to fit within the biologically-inspired design parameters. Perhaps more importantly, the tolerance that neural systems have against faults must somehow be migrated into the non-neural domain.

The rest of this thesis addresses these two points, by first presenting a set of algorithms that can detect faults and configure the networks to ignore them in Chapter 5, and then by presenting an example non-neural application in Chapter 6. However, before diving into descriptions of this work, it must be clear how the SpiNNaker system itself works

and how neural applications are structured to take advantage of it. This is presented next in Chapter 3.

Chapter 3

SpiNNaker

3.1 Hardware Composition

3.1.1 The SpiNNaker Chip

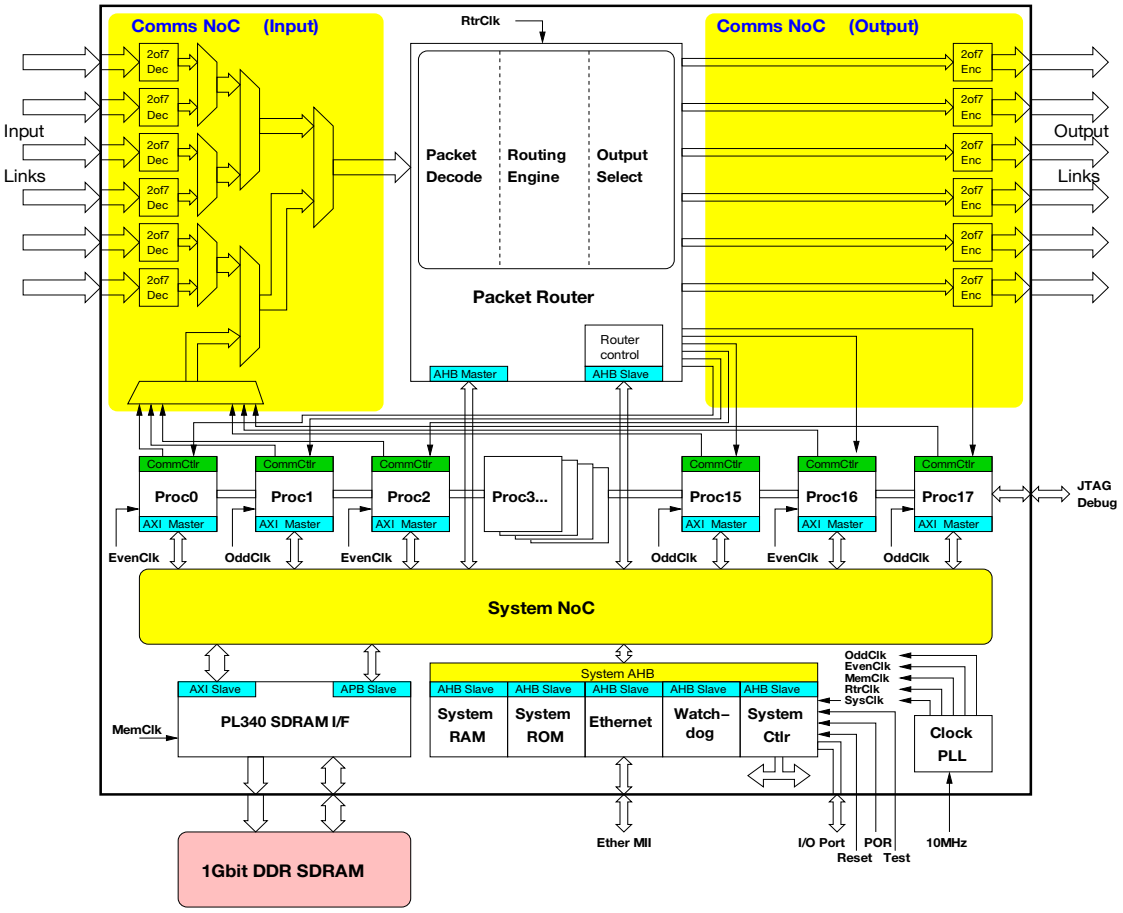


Figure 3.1: Detailed block diagram of the SpiNNaker chip (taken from the datasheet [115]).

The SpiNNaker chip architecture [6, 35, 116], shown in Figure 3.1, contains 18 ARM968 processors embedded in two separate network-on-chip (NoC) structures. One, the system NoC, connects the processors to the various on-chip peripherals, and the second, the communications NoC, connects the processors to each other and to six bidirectional ports used for inter-chip communications [117]. The communications system in SpiNNaker is central to its operation is will be explained in section 3.1.2.

A system NoC is used in place of a more conventional system bus architecture because there are many processors sharing many resources. In typical bus-based systems with multiple masters, an arbiter decides which master to grant access to the bus. Once access has been granted, this master has *exclusive* access, thereby prohibiting any other master from accessing any part of it, even if the simultaneous operations would not collide. For small systems with relatively few masters, this is an acceptable cost. However, for systems such as SpiNNaker with a large number of masters, this is not acceptable [118].

Instead, SpiNNaker chips use a Silistix CHAIN packet-switched NoC [119] as shown in Figure 3.2 [117]. Multiple masters can access devices simultaneously through this NoC without requiring exclusive access. Operations that would conflict (for example, two processors attempting to write to the same SDRAM address) are naturally arbitrated by the NoC itself without either processor needing to act. From the perspective of each processor, the system NoC is indistinguishable from a conventional bus and forms the node-local part of the memory map shown in Table 3.1.

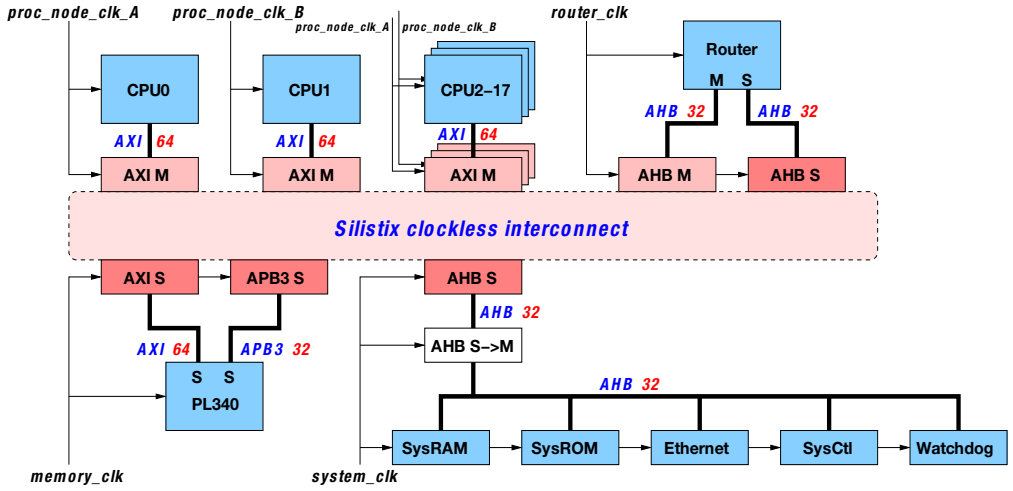


Figure 3.2: Structure of the System NoC [117].

Each processor block in Figures 3.1 and 3.2 is a small sub-system that includes additional core-local resources, as shown in Figure 3.3, which form the remainder of the memory map in Table 3.1. Within this block, the processor has uncontested control of the AHB1 bus that connects it to the core-local peripherals.

Access to node-local resources is achieved through the DMA controller and the associated 'CHAIN gateway.' Both this and the communications controller convert the synchronous

Offset	Usage	Buffered
0x00000000	ITCM	n/a
0x00008000	Not used	n/a
0x00400000	DTCM	n/a
0x00410000	Not used	n/a
0x10000000	Local peripherals - comms, counter, VIC, DMA	mixture
0x50000000	Bus error	n/a
0x60000000	SDRAM	yes
0x70000000	SDRAM	no
0x80000000	Bus error	n/a
0xe0000000	NoC peripherals - router, controller, watchdog, ethernet	yes
0xe5000000	System RAM	yes
0xe6000000	System ROM	yes
0xe7000000	Bus error	n/a
0xf0000000	NoC peripherals	no
0xf5000000	System RAM	no
0xf6000000	System ROM	no
0xf7000000	Bus error	n/a
0xff000000	Boot area	no

Table 3.1: System memory map for each SpiNNaker chip [35]. Rows that are shaded indicate core-local resources, while the reset are node-local.

bus into the self-timed packet-switched traffic for the two NoCs. Essentially, each processor, each node-local peripheral, and the router are 'islands' of synchronous sequential logic embedded in an asynchronous connection fabric provided by CHAIN. Details on the router and how the NoCs work are provided in section 3.1.2.

Whilst CHAIN is capable of having multiple transactions occurring simultaneously, it clearly cannot allow multiple masters simultaneous access to the same physical resource. The resources most likely to cause this contention are the system RAM and the SDRAM. As mentioned earlier, SDRAMs already carry a high latency compared to modern processors, and with so many masters contesting for this resource, access times could be further slowed. However, this is one of the reasons the DMA controller is connected to the CHAIN gateway: to allow it to be used for latency-hiding when accessing shared

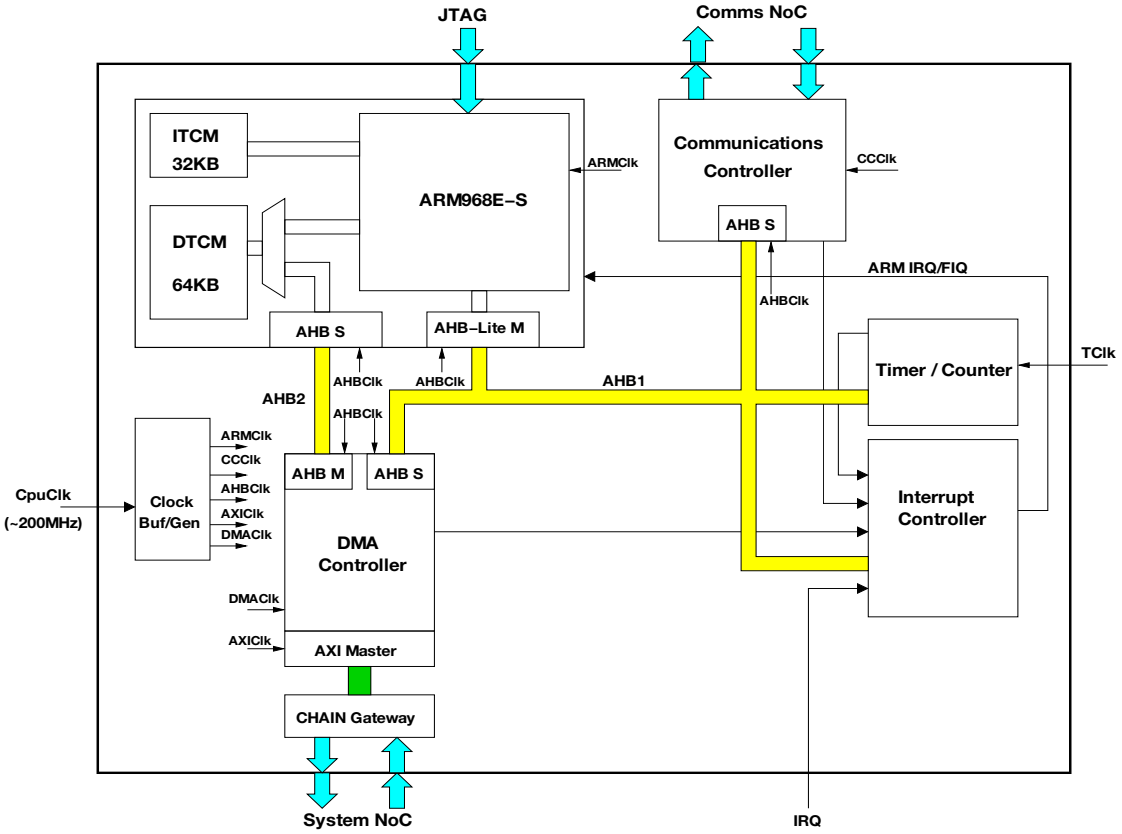


Figure 3.3: Detailed block diagram of a SpiNNaker core subsystem [117].

resources [35]. Once the data has been written into the DTCM (data tightly-coupled memory), the DMA controller can raise an interrupt in the processor, allowing it to resume the operation that requested the data.

The primary purpose the SDRAM serves is for essentially increasing the accessible memory of each processor in the chip. Parameters and data specific to the problem being solved at hand (e.g., synapse parameters) are stored in the SDRAM, but will typically only be accessed by the individual processor hosting that part of the problem (similar to the SPMD model). These reasons combine to rule the SDRAM out for shared memory communications. In fact, SpiNNaker differs from most platforms in this instance because sending a communications packet between cores (even on the same die) is faster. However, for initial configuration, debug, and low-priority message traffic, shared-memory communications using the system RAM (*not* the SDRAM) are used for the final delivery stage.

Before moving on to describe the networking infrastructure of the chip, it is worth noting that each core has its own counter/timer. The importance of this will become apparent in section 3.3 when the structure of neural software is introduced. Two separate counters are included, each with separate interrupt request (IRQ) lines connected to the core interrupt controller.

3.1.2 Network Infrastructure

3.1.2.1 Communications NoC

The system NoC has multiple devices to route between, whereas the communications NoC essentially serves as a multi-layered input tree for gathering packets and an output stage for transmitting them [90]. Figure 3.4 shows this topology for the NoC with the router placed between them. Merge trees placed throughout the input stage collect narrow bit-streams operating at high effective clock-rates into wider ones operating at slow clocks. At the lowest stage incident upon the router, the bit-streams are 72 bits wide, as large as the largest possible SpiNNaker packet.

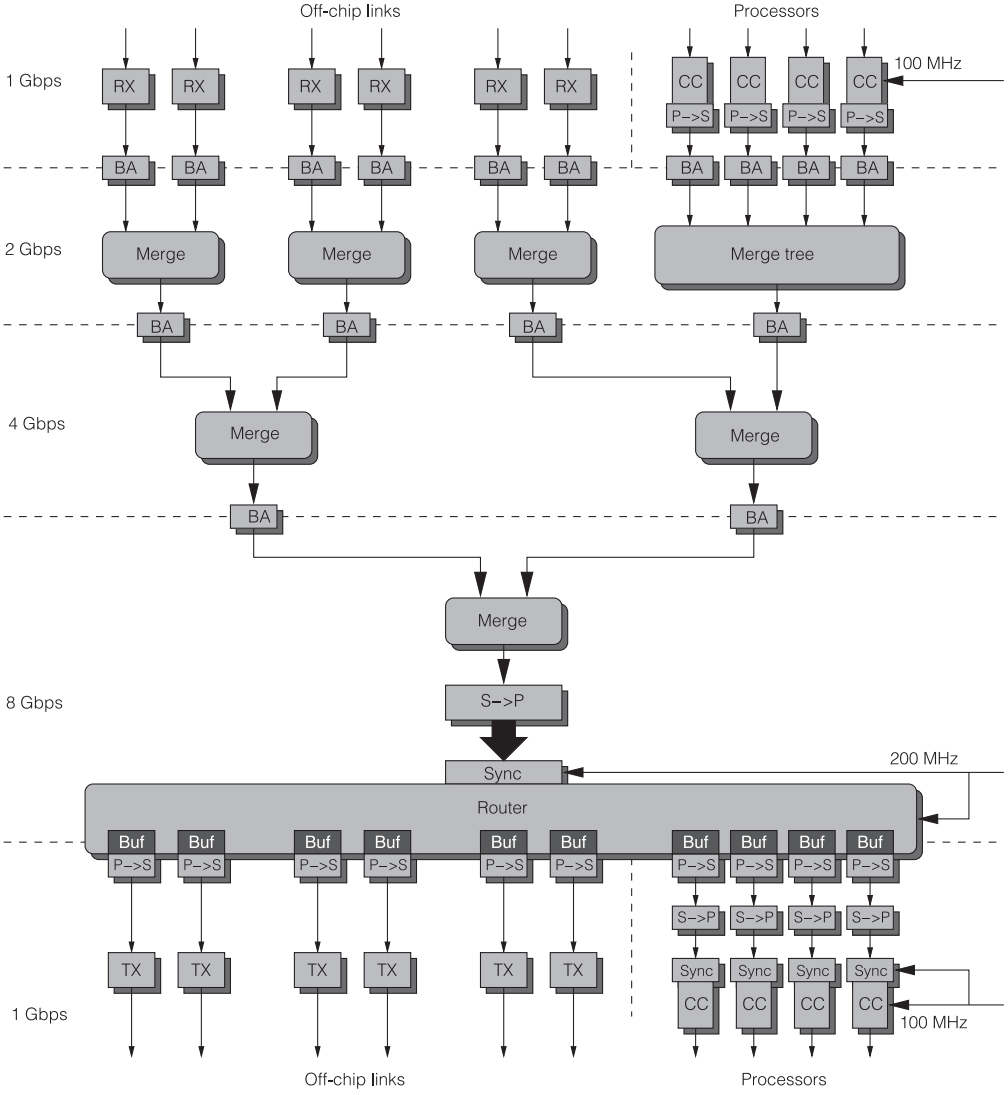


Figure 3.4: Structure of the Communications NoC showing merge trees, bandwidth aggregators (BA), serial-to-parallel converters (S->P), and parallel-to-serial converters (P->S) [90].

Both the system and communications NoC use CHAIN 1-of-5 return-to-zero signalling [119]. CHAIN is a self-timed system requiring no clock signal to drive the traffic forward. This allows the network to degrade performance naturally depending on how quickly packets are being removed from it. If a device becomes overwhelmed, the network automatically adjusts to this back-pressure without needing any active control. Devices transmitting data into the network will almost certainly be synchronous systems, hence they will need to make active decisions about whether to transmit or not.

The signalling used by these NoCs is therefore *delay insensitive* (DI) and conforms to the more general family of m-of-n DI codes, meaning that each symbol has exactly m bits of an n bit data-line set for each symbol. In the general case, an m-of-n code can transmit ${}_nC_m$ bits per symbol [120]. CHAIN's 1-of-5 encoding therefore includes 5 discrete states that a symbol may take: `0b00`–`0b11`, and end-of-packet (EOP); i.e., each CHAIN symbol carries 2 bits of data using five data-wires.

A sixth *acknowledgement* wire feeds from the receiver to the transmitter to accept the symbol with which it has been presented. A complete CHAIN symbol transmission consists of the following four phases [119]:

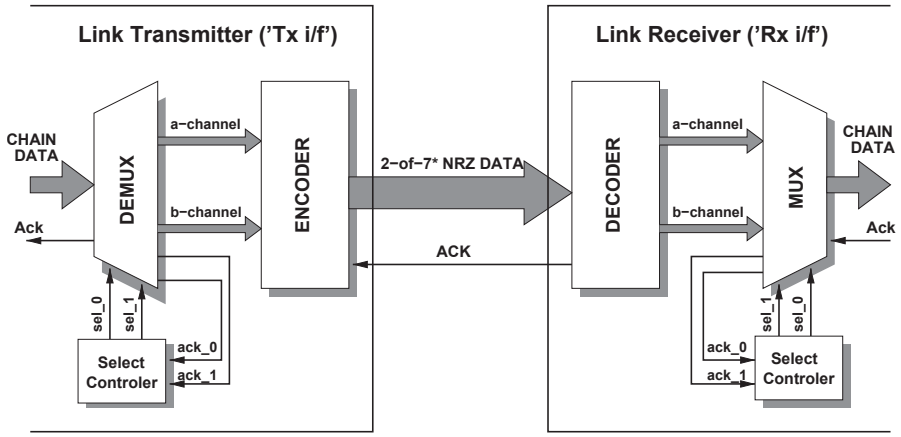
1. One of the five data-wires is driven high to represent the appropriate symbol.
2. The receiver drives the acknowledgement wire high to accept the symbol.
3. All data wires are driven low to restore the bus to its idle state.
4. The acknowledgement wire is driven low by the receiver to indicate to the transmitter that the next symbol can be accepted.

The procedure is therefore *self-timed*, because the transmission cannot continue without both the transmitter and receiver agreeing, and *return-to-zero* (RTZ), because both the data and acknowledgement wires are reset after the transmission completes. These four phases lead to simple synchronisation logic between multiple CHAIN components operating in parallel to increase the number of bits transmitted simultaneously.

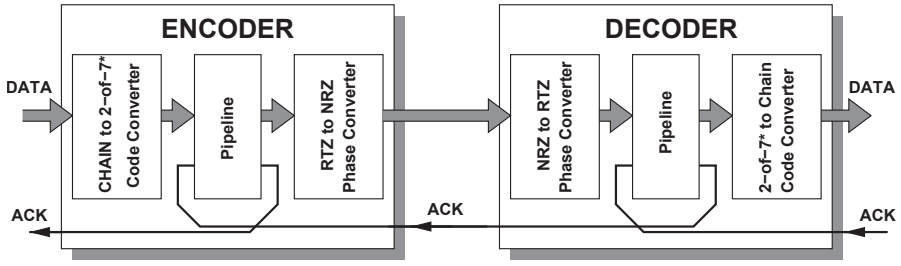
This delay-insensitive principle also continues across the boundary between chips. However, pins on a package are a strictly limited resource and have electrical properties that bound their switching frequency compared to wires in the metal layers of the chip. Inter-chip communications must therefore carry more bits-per-symbol and, ideally, at a lower symbol rate.

Inter-chip communications use a 2-of-7 non-return-to-zero (NRZ) DI scheme instead and have ${}_7C_2 = 21$ states for each symbol. Sixteen of these states are used for the binary values `0b0000`–`0b1111` and one is used for EOP. NRZ means that fewer phases are required to convey a symbol. Data is presented by a transmitter *toggling* any two of the

seven data lines, and data is accepted by the receiver by *toggling* the acknowledgement line. At any single point in time, the logic values of the bus do not necessarily convey the symbol being transmitted. Despite this complication, the block diagrams given in Figure 3.5 operate at a bandwidth of (nominally) 250Mbps [91]. As not all of the 21 available symbols are used, it is considered an *incomplete* 2-of-7 code, which is usually denoted by 2-of-7*.



(a) CHAIN to 2-of-7 physical layer inter-chip connection.



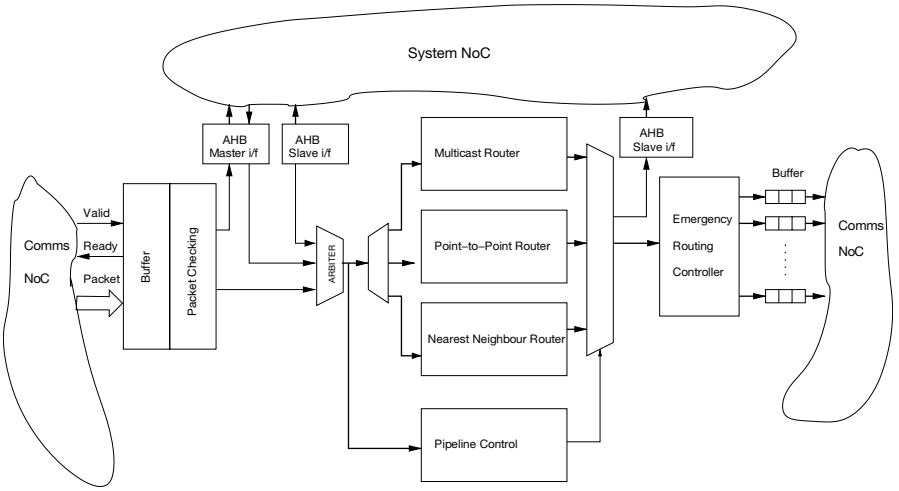
(b) Encoding/decoding pipeliens to convert between CHAIN 1-of-4 and inter-chip 2-of-7.

Figure 3.5: SpiNNaker chip-to-chip communications [91].

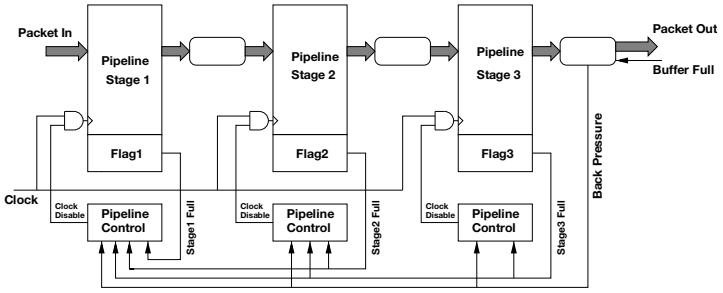
3.1.2.2 Routing

Figure 3.6a shows an exploded view of the 'Router' block in Figure 3.4. The router block is a synchronous system unlike the two NoCs to which it is connected. Back-pressure can be applied to it just like it can apply back-pressure to other systems. To tolerate this, each stage is pipelined with specific feedback connections to pause the earlier stages accordingly (see Figure 3.6b) [121].

A forward-stage of the router checks that the parity of the incident packet is correct, and if this check fails, the packet is simply dropped. Once past this gate, packets enter one of three specific routing stages depending on the packet type, the specifics of which will be explained in the next section.



(a) Block diagram of the SpiNNaker router.



(b) Global stall control implementation.

Figure 3.6: SpiNNaker router implementation [121].

The final stage before output into the communications NoC is to perform *emergency routing* if required. SpiNNaker chips are connected together in a triangular pattern that will be described in section 3.1.3. This allows the router to divert traffic around a congested link by sending them through a triangular neighbour instead. Traffic sent this way is marked as having been emergency-routed so that the neighbour can forward the packet onto its original target.

3.1.2.3 Packet Types

Each SpiNNaker node is assigned a unique 16 bit identifier (hard-limiting the system size to 2^{16} nodes), and each core within a node has as a 5 bit index; the details of how both of these are assigned are given in section 3.2.1. Index zero is reserved for the *monitor* core which performs all node-level functions, leaving the indices 1–17 for the remaining *worker* cores which host application code.

SpiNNaker packets take the form shown in Figure 3.7 and may be 40 or 72 bits in size. The first 8 bits comprise the header which the router uses to select the appropriate processing path (and to resolve diversions introduced by emergency routing) in Figure 3.6a [35].

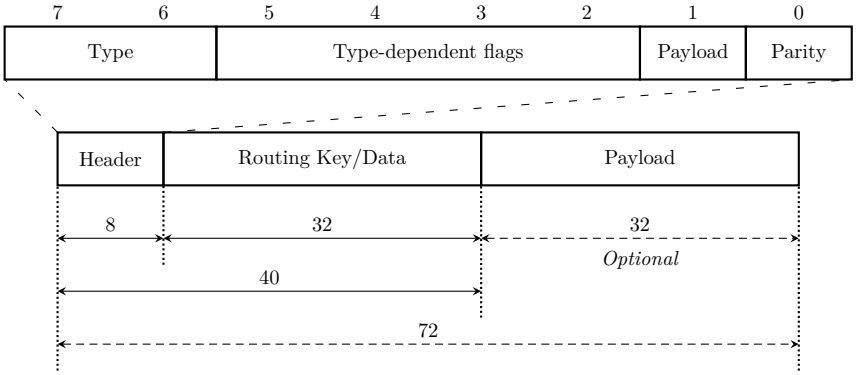


Figure 3.7: SpiNNaker general packet format.

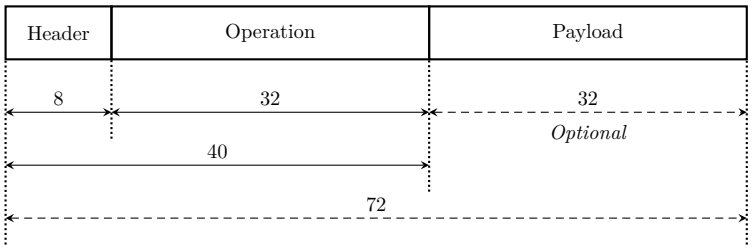


Figure 3.8: Nearest-neighbour (NN) packet.

Packets may take one of four types: nearest-neighbour (NN), point-to-point (P2P), multicast (MC), or fixed-route (FR). NN packets (Figure 3.8) are the simplest of this taxonomy by only supporting transmission between immediately neighbouring nodes. However, this also means that they require no routing information and are hence *always* usable. The `opcode` field permits inter-chip operations such as the *peek* and *poke* (i.e., read and write) of the memory spaces of neighbouring nodes. NN packets are essential for machine initialisation, and come in two sub-types: *normal* for the general exchange of data, and *peek/poke* to allow nodes to access the system NoC resources of their neighbours.

P2P packets (Figure 3.9) allow any two nodes to communicate without requiring software intervention. Once the P2P tables in the router of each node have been appropriately configured, their transmission is entirely brokered by the router of each node. P2P tables effectively form a map of target node ID to router output port, so, locally, the only piece of information each node needs is the identity of the port through which to forward a P2P for it to (eventually) reach its destination. They are an entirely decentralised form of communication that obviously need careful configuration (explained in Section 3.2). These packets form the basis of targeted system-level traffic such as debug outputs and data downloads from SpiNNaker, or targeted commands and uploads to SpiNNaker.

MC packets (Figure 3.10) form the basis of application-level traffic and are the only packet type capable of targeting specific cores within a node. NN and P2P packets incident to a router raise an interrupt that any core in that node may service, whereas MC packets will raise an interrupt in the desired target core. MC packets also require

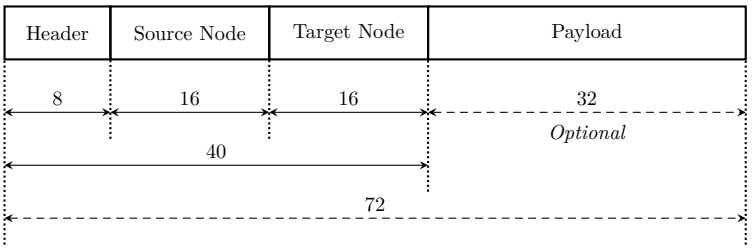


Figure 3.9: Point-to-point (P2P) packet.

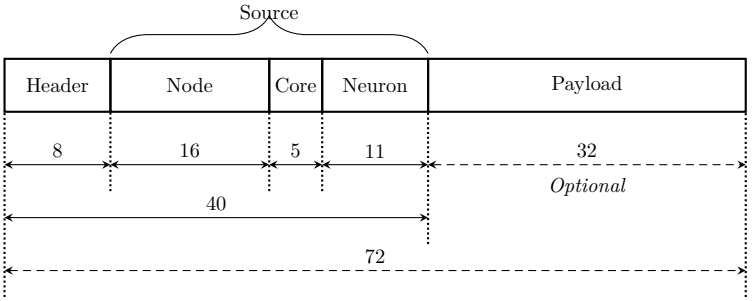


Figure 3.10: Multicast (MC) packet.

that specific information is written into the router, but into a special content-addressable memory (CAM) called the **MC table** [121].

This CAM (see Figure 3.11) includes a *mask* and a *value* for each of the 1,024 entries it contains which can be compared against the *source address* of the inbound MC packet. This is effectively a tri-state bitwise OR between the address and *all entries in the CAM simultaneously*. Each of these comparisons produces a result that is stored in an output register, the highest valued of which is selected. The index of the bit is decoded into an address in an output RAM which contains two n-hot data fields where each bit represents a valid output on the communications NoC; i.e., any of the 18 cores and/or any of the 6 neighbours. MC packets can therefore be duplicated very efficiently and entirely by hardware once the routers have been appropriately configured.

As MC packets are routed by their *source* address rather than to a particular destination as would be the case in more typical architectures, the addresses follow the address-event representation (AER) [122]. This allows neural spikes (i.e., the communication between elements of a neural simulation) to be routed extremely efficiently despite potentially having very large fan-outs.

FR packets Figure 3.12) are the fourth and final type and can be thought of as a special case of an MC packet with a CAM with only a single entry (i.e., all packets match). This "CAM" is actually just a single register in the router of the same format as the entries in the output RAM of the MC router. This allows FR packets to be routed to a single destination efficiently, which is useful for supporting debug print-statement outputs to a host computer.

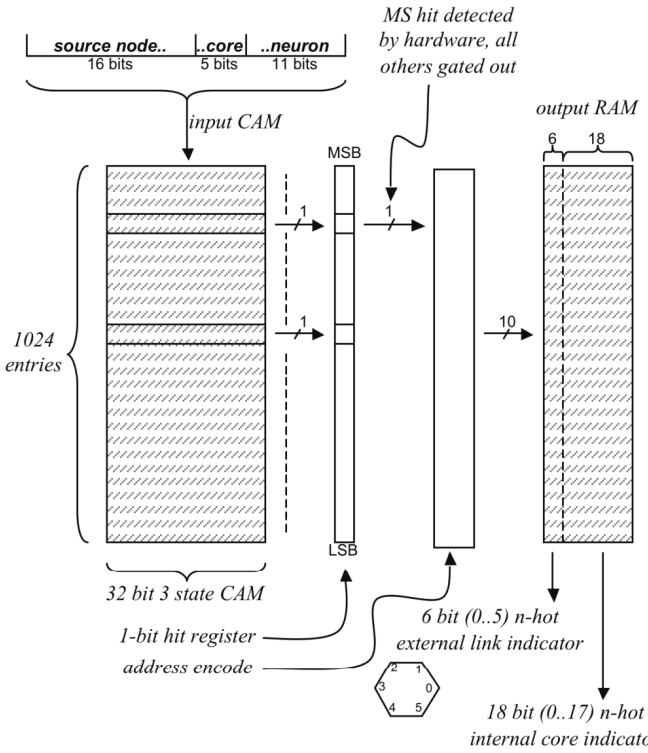


Figure 3.11: Mutlicast router content-addressable memory (CAM) [35].

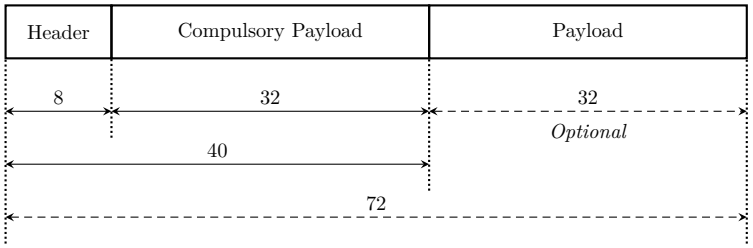


Figure 3.12: Fixed-route (FR) packet.

3.1.3 Multi-chip systems

Chips in SpiNNaker form a 2D torus, shown in Figure 3.13, which is homogeneous and isotropic. Clearly the chips must be mounted on printed circuit-boards to construct a system that is useful. Inter-board traffic is brokered by FPGAs on each board as shown in Figure 3.14 [2]. Additional details about the inter-board connections, including extensions to support plug'n'play peripheral I/O devices, can be found in Dugan, Brown, Reeve, *et al.* [2] and Dugan, Brown, and Reeve [123].

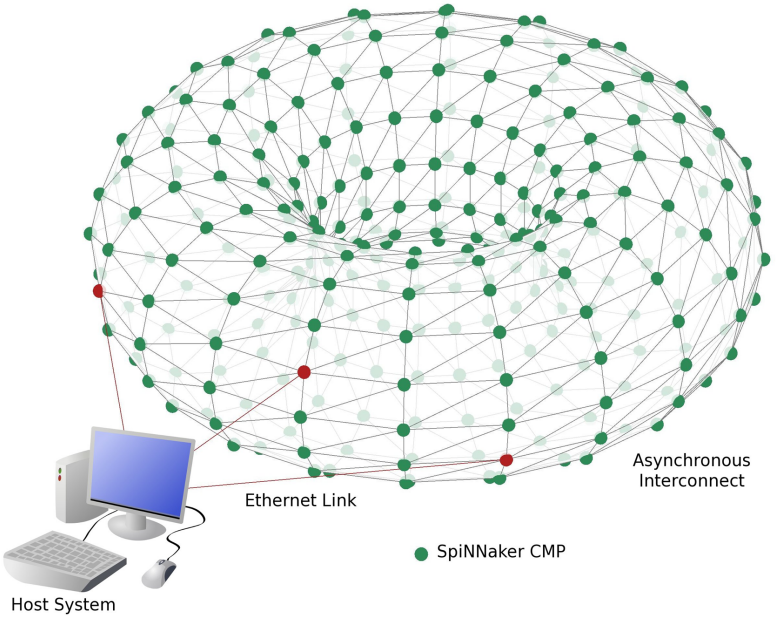


Figure 3.13: Connection diagram for a complete multi-chip SpiNNaker system showing external host connections.

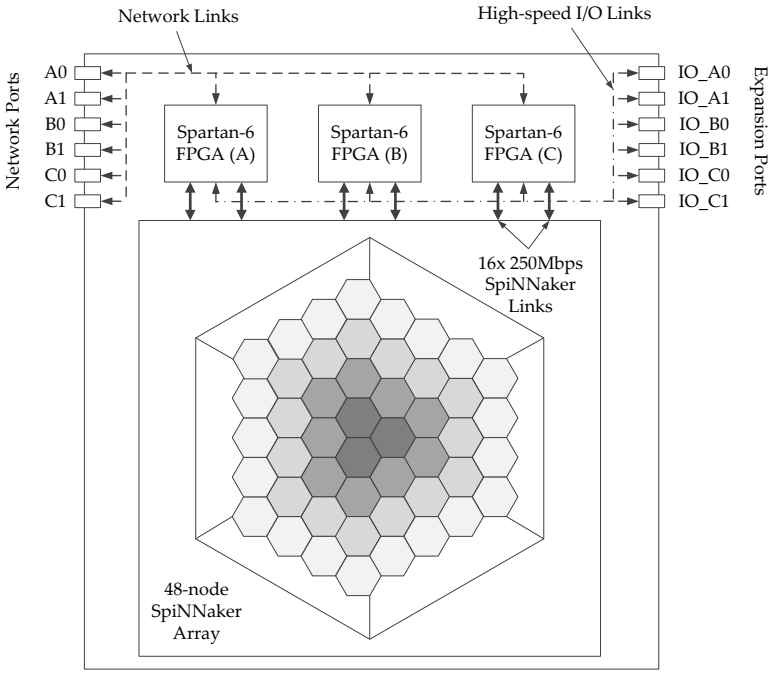


Figure 3.14: Schematic view of a SpiNNaker-103 board showing inter-board connections [2].

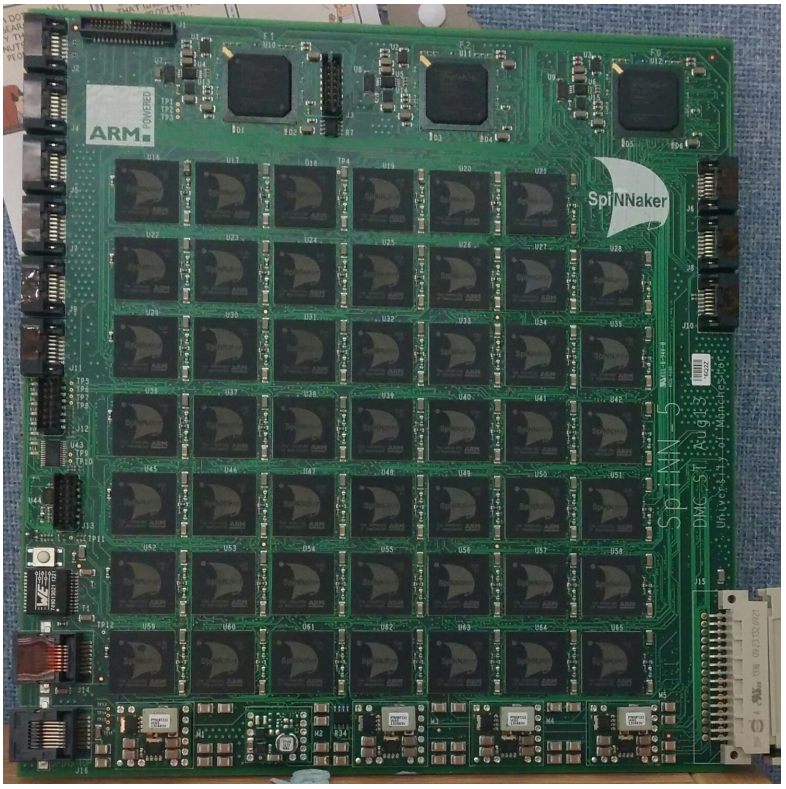


Figure 3.15: A SpiNNaker-103 system, containing 48 chips and 864 processors.

To date, several SpiNNaker systems have been built of varying sizes:

- Figure 3.15 shows a SpiNNaker-103 (shown schematically in Figure 3.14) containing 864 processors distributed over 48 nodes.
- Figure 3.16 shows a SpiNNaker-104 containing 20,736 processors over 1,152 nodes, forming the network pattern shown in Figure 3.17. Each large hexagon labelled Sn represents a SpiNNaker-103 system, with the hexagonal pattern shown in Figure 3.14.
- Figure 3.18 shows five SpiNNaker-105 machines containing 103,680 processors each, or 518,400 when combined.

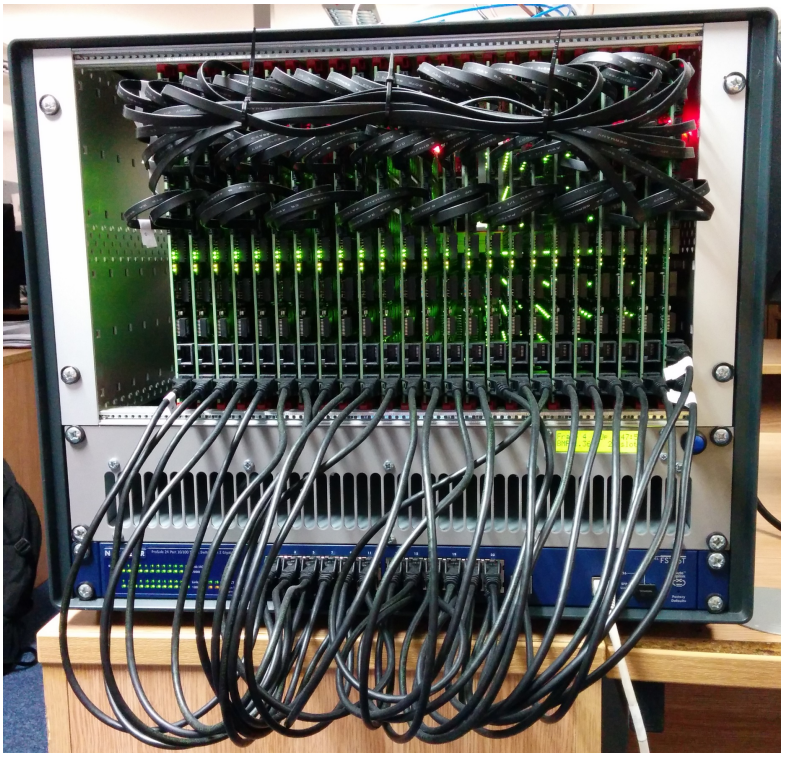


Figure 3.16: A SpiNNaker-104 system comprised of 24 SpiNNaker-103 boards for a total of 1,152 chips and 20,736 processors.

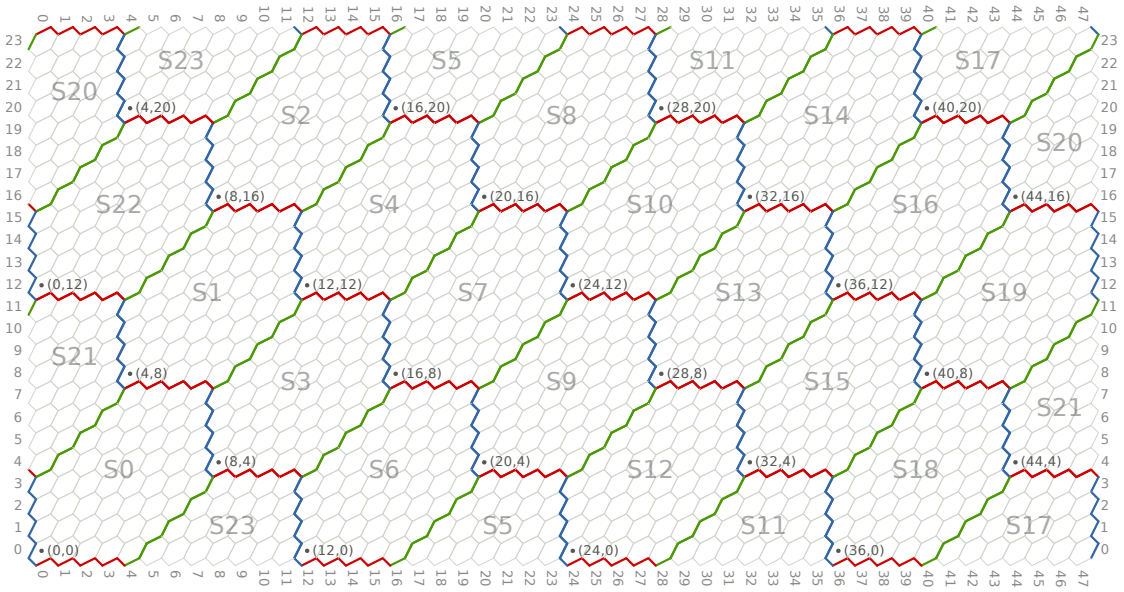


Figure 3.17: Connection pattern for a SpiNN-104 machine composed from 24 SpiNN-103 boards [124].

Figure 3.18: Five SpiNNaker-105 cabinets forming the largest SpiNNaker system assembled to date. Each SpiNNaker-105 is built from five SpiNNaker-104 systems for a total of 5,760 chips and 103,680 processors. The system in this picture therefore contains 28,800 chips and 518,400 processors.

3.2 Loading Software

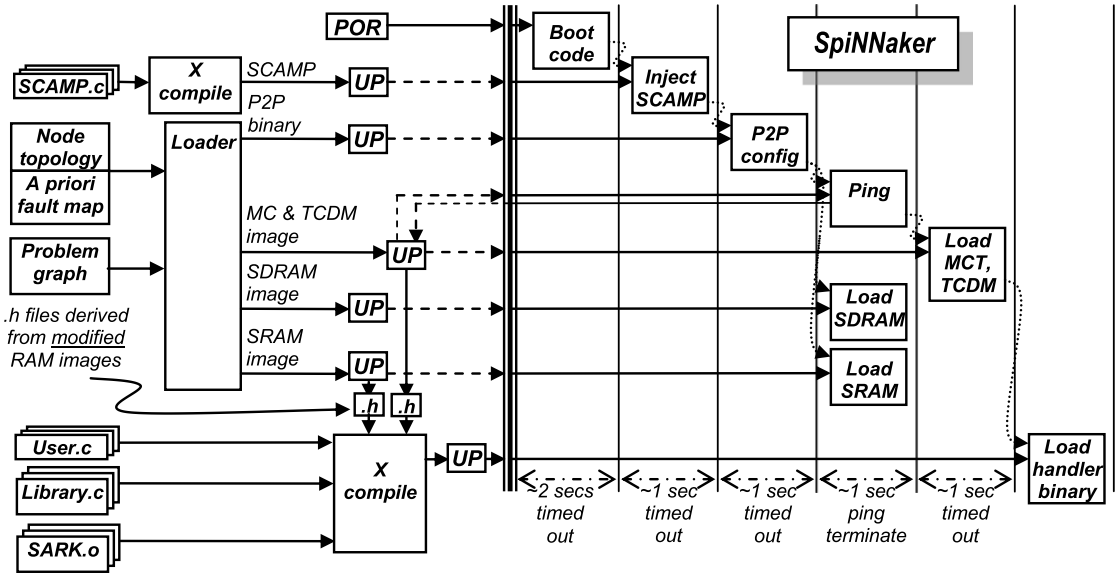


Figure 3.19: Complete SpiNNaker boot sequence from power-up to starting an application [6].

SpiNNaker has no persistent memory and hence must perform a complete reinitialisation after every power cycle. Figure 3.19 shows the boot sequence as both a tool and a

temporal flow [6]. The process is clearly non-trivial, but can be broken into two distinct procedures. First the machine must be taken from a cold boot to a point where it can accept commands: the boot process described in section 3.2.1 (and in greater detailed in Appendix B). Secondly, user applications can be uploaded so that a simulation can take place, as described in section 3.2.2.

SpiNNaker differs from conventional parallel machines in that applications are decomposed into a fine-grained graphical description called a *problem graph*. Each node of this graph (a *problem device*) describes a very specific part of the overall problem as a set of *event handlers*, which react to the reception of messages that follow the problem graph edges and the firing of a periodic timer tick. Mapping the problem graph to the physical hardware (see Figure 4.4) is a non-trivial problem which is briefly explained in Chapter 4.

3.2.1 Boot Process

Immediately following power-up, every core and hardware resource is reset by the node-local Power-On Reset (POR) controller, which causes every core in SpiNNaker to execute the boot code stored in the system ROM (memory mapped, see Figure 3.1 and Table 3.1). This boot program performs a brief Power-On Self-Test (POST) of the core-local peripherals in Figure 3.3, and then moves on to assign the *virtual core identifiers*. This is achieved by each processor accessing special registers in the node-local system controller which serve as arbiters for each assignable ID. These registers also fully encapsulate the virtual-to-physical core map local to the node, and it follows that, as this process is non-deterministic, that physical core N is not necessarily virtual core N.

Once virtual core IDs have been assigned (the only IDs of any use from this point forward), a natural hierarchy of processors has been established. Core zero is always the *monitor* core which assumes the role of the master core in the node. Cores 1–16 are available as worker cores to the application software that will be uploaded later, and core 17 serves as a spare in case one of the workers faults. The monitor of each node now begins listening for boot commands being carried over NN traffic. At least one node in the system will have a working Ethernet connection, and the monitor in this node also begins listening for UDP packets (over Ethernet) carrying boot commands. This includes the 'boot code' block in Figure 3.19.

The next phase is to program the machine with system software which provides a command interface and handles the execution of applications. A conventional computer serving as the *host* begins the boot process by uploading a cross-compiled SCAMP (SpiNNaker Control and Monitoring Program) binary using a boot protocol carried by UDP packets. For the sake of simplicity, it will be assumed that this will be sent to a

single Ethernet-connected node even if there are several with this capability. This node will be considered the 'root' of the system.

Once the SCAMP image has been completed received by the root node, the boot program will cede control of the hardware [125]. SCAMP then immediately copies itself to all other cores on the node, and then begins copying itself to all immediate neighbours using NN packets. These nodes then perform the same sequence of events, causing SCAMP to *flood-fill* to every core in the system [126]. Completion of this process cannot be detected because the size of the machine is not yet known (the same is true for the boot processor), hence a timeout is used as shown in Figure 3.19.

To conclude this phase, the root-node monitor begins listening for SpiNNaker Command Protocol (SCP) command messages being carried in UDP packets. All other nodes in the system also listen for SCP messages being transmitted over the P2P infrastructure instead. However, at this point in the sequence, the P2P tables are not configured, hence the third and final phase of the boot sequence.

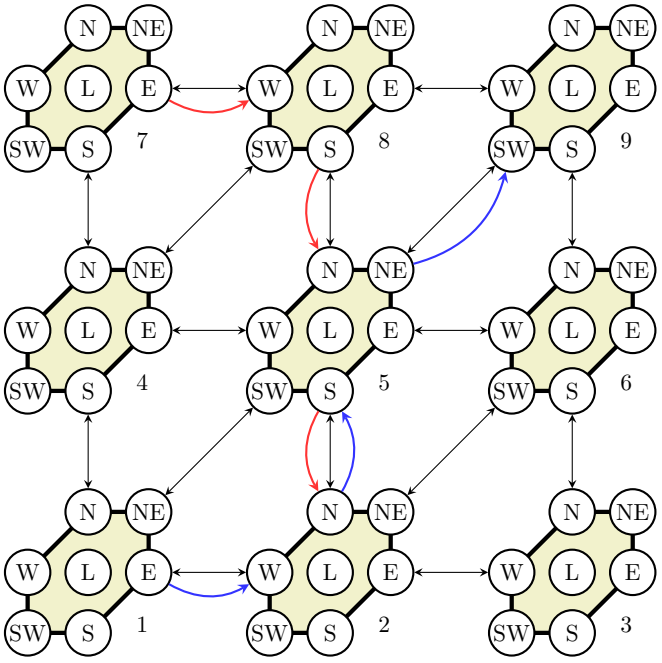


Figure 3.20: Example P2P routes on a 3×3 SpiNNaker grid.

Figure 3.20 shows an example of a 9 node SpiNNaker fragment connected as a 3×3 grid. The P2P tables present in each node form a map showing which port a node must send traffic through to eventually get to the destination. A node does not know (or need to know) the exact route that this message will take, just that if a message is forwarded through a particular port it will eventually reach its destination.

Figure 3.21 shows an example of how *all* the tables of the Figure 3.20 system might be configured. Each column is the complete P2P table of node N, and each row is the port required to reach a particularly destination. In both figures L represents the *local*

virtual port which will arrive at the monitor of that node. Several entries in Figure 3.21 are shaded either red or blue, and these entries completely describe the red and blue example paths in Figure 3.20.

Target Node \ Node	1	2	3	4	5	6	7	8	9
1	L	W	W	S	SW	SW	S	SW	SW
2	E	L	W	E	S	SW	E	S	SW
3	E	E	L	E	S	S	S	W	S
4	N	W	W	L	W	W	S	SW	W
5	NE	N	W	E	L	W	S	S	SW
6	NE	NE	N	E	E	L	S	S	S
7	N	W	W	N	W	W	L	E	W
8	N	N	W	NE	N	W	E	L	W
9	E	N	N	NE	NE	N	E	E	L

Figure 3.21: Example P2P configuration for the system of Figure 3.20. The red and blue shading highlights entries relevant to the red and blue routes respectively. The yellow shading highlights node-local delivery.

Building the P2P tables is a two-step process initiated by the external host machine issuing the `p2pc` command with the system dimensions. The first step is for each node to assign itself a unique identifier; the system dimensions are used here to correctly calculate an identifier. Each node (starting from the root) propagates its ID along with the machine dimensions to its north, north-eastern, and eastern neighbours. This is another flood-fill process and completion cannot be detected. After a suitable timeout of a about 10ms, each node then broadcasts its ID to *all* neighbours which allows each core to populate its P2P table. The timeout in Figure 3.19 treats both of these steps as a single operation. This is the existing approach which is well-suited for use on a SpiNNaker toroid, but it cannot configure machines of arbitrary topology, which is addressed by the contributions given in Chapter 5.

Once this is complete, all nodes in the system will: be running SCAMP on every core, have a unique 16 bit identifier, have a valid P2P table, and be awaiting further instructions from SCP commands.

3.2.2 Uploading Applications

There are three software tools in Figure 3.19 which each run on the host machine:

1. The *Loader* takes a *problem graph* (introduced in section 3.3) and produces from it sets of MC routing tables for the CAM of Figure 3.11 and node-local and core-local data.

2. The commercial *cross-compiler* (X compile) produces binaries for the ARM968 processors from 'C' code and various support libraries.

3. The *Uploader* which handles all physical interactions with the SpiNNaker machine (including the boot and P2P configuration described previously) and was specifically written as part of the work in this thesis. A description of this tool is given in reference [127].

An application in SpiNNaker is composed of three types of data: 1) programs (usually homogeneous) for each core, 2) data for each program to process, and 3) MC routing information providing means for each program (or component thereof) to communicate. The Uploader supports two mechanisms: a *flood-fill* (as described earlier) which can be used for homogeneous programs, and a *directed load* which can be used for node- and core-specific data, MC routing information, and heterogeneous programs.

Before any core-specific information can be uploaded, the Uploader performs a *core liveness survey* (the 'Ping' block in Figure 3.19) to determine which worker cores are capable of accepting programs and data. This is required because a core may pass the POST and obtain a low-order ID and then fail after this point. Despite the rarity of this actually happening, it must still be dealt with appropriately for applications to function correctly. Node-specific data (the 'Load SDRAM' and 'Load SRAM' blocks of Figure 3.19) can be uploaded *whilst this is taking place*.

Much of the application logic of a SpiNNaker program is contained within the MC routing tables because it directs traffic to the appropriate worker cores. This information is generated off-line before the upload has even started (the process is a complex task that can take many hours for large problems [128]), and it is vital that the offline model of the system conforms exactly to the hardware state of the target machine. The Uploader can use the results of the core-liveness survey to dynamically remap the MC destinations to account for faulty cores. Clearly remaps can only occur if the following conditions hold true:

$$N_w \geq N_r \text{ and } N_t - N_w > 0, \tag{3.1}$$

where N_t is total number of workers in the node, N_w is the number of worker cores that are functioning, and N_r is the number of works required by an application. Typically $N_t - N_w = 1$ but to increase manufacturing yield, cores with a single processor fault are also accepted, hence $N_t - N_w = 0$ for these nodes. Additionally, core-local data must

be migrated according to the remaps. The 'Load MCT, TCDM' block of Figure 3.19 represents this behaviour and shows that this occurs *after* the survey.

The Uploader generates a set of 'C' header files as it is processing the data for upload. These files contain various memory offsets and base-addresses, as well as constants that refer back to the original problem specification that was given to the Loader. An off-line cross-compilation builds a (usually homogeneous) application binary from these headers, a support library to correctly unpack the uploaded data structures, and the application code itself. Once this binary exists, the Uploader performs either a flood-fill (for SPMD applications) or a directed load (for MPMD applications) taking into account any core remaps.

At this point, the machine is ready to begin the simulation by issuing the appropriate SCP command. The Uploader can issue this automatically, or can stop after this point to allow the user to interrogate the state of the machine before it starts.

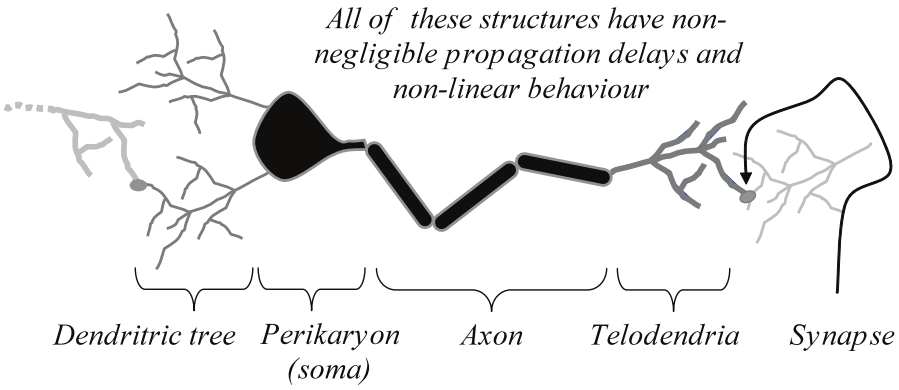
3.3 Neural Simulation

3.3.1 Neural Systems

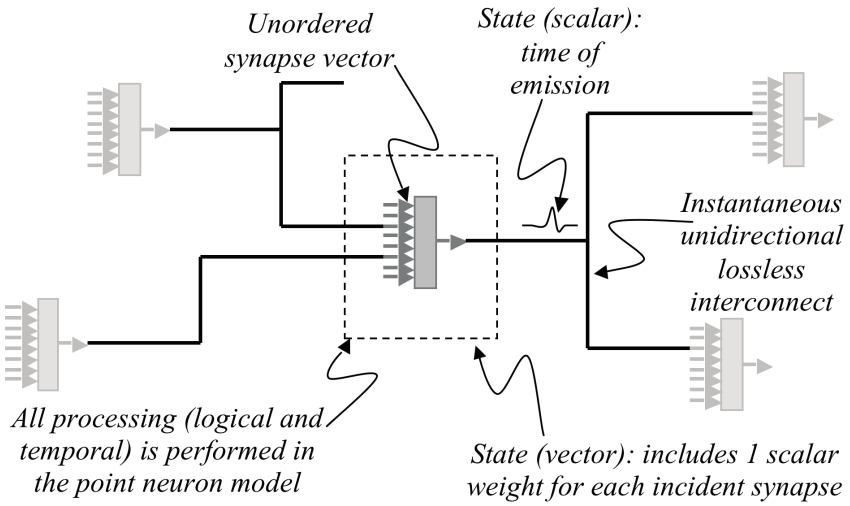
Neural systems exhibit massive parallelism and incredible fault-tolerance when it comes to solving problems, and, perhaps more impressively, they consume much less power (about 25W for a human brain [35]) than even a desktop computer system (several hundred watts). The fundamental components of a neural system are shown in Figure 3.22a. The neural simulators described in section 2.2 (including SpiNNaker) make use of the *point neuron model* where the particulars of the dendritic tree are lumped in with that of the soma, and the synapse models the physical interconnection. NEURON is an exception to this generalisation because axon (or cable) properties are modelled very accurately [94].

Axons behave similarly to transmission lines in electrical systems and carry signals between connected somata. Synapses serve as gates or junctions that perturb the incoming signal before it reaches the target soma. In real biological systems, most of these signals are short pulses known as *action potentials* which are easily modelled as a narrow voltage spike [35]. SpiNNaker makes use of this approximation, which is one of the design reasons that the payload on MC packets is optional; the presence of a packet *is* the data, much like the presence of a spike *is* the signal in biological systems, because it conveys the wall-clock arrival time.

A SpiNNaker application, therefore can be viewed as a graph, $G = (V, E)$, where the vertices are code fragments that implement a specific neuron model, and the edges are fragments of code that implement synapse models. Figure 3.22b shows this structure;



(a) Schematic view of a biological neuron.



(b) A network of modelled neurons as they appear in a SpiNNaker application.

Figure 3.22: Visual representation of how a biological network is mapped into a neural network on SpiNNaker [35].

the 'unordered synapse vector' is an array of all post-synaptic signals, and the shaded grey box is the neuron model itself, which may produce spikes. A core can host between one and approximately 10^3 neuron models, and up to about 10^6 synapse models, of varying levels of biological detail.

3.3.2 Modelling with an Event-based Architecture

SpiNNaker is an interrupt-driven system, so it follows that programs are structured as a set of *handlers*. Figure 3.23 shows the process that the SpiNNaker Application Run-time Kernel (SARK) uses to invoke handlers in response to interrupt-events triggered by either a packet arrival or the timer tick. SARK is shown on Figure 3.19 as one of the libraries that is linked into application code. In addition to implementing the interrupt delivery behaviour, it overwrites SCAMP on the worker cores (Appendix B), thus allowing SCP

commands to be handled as before, whilst also supporting the user-defined application code.

Application code can implement specific commands to help with the simulation, the most common example of this being to provide a means of extracting results. Once an application has finished, SARK handles upload and execution commands identically to SCAMP so that another simulation can be performed without requiring a reboot of SpiNNaker.

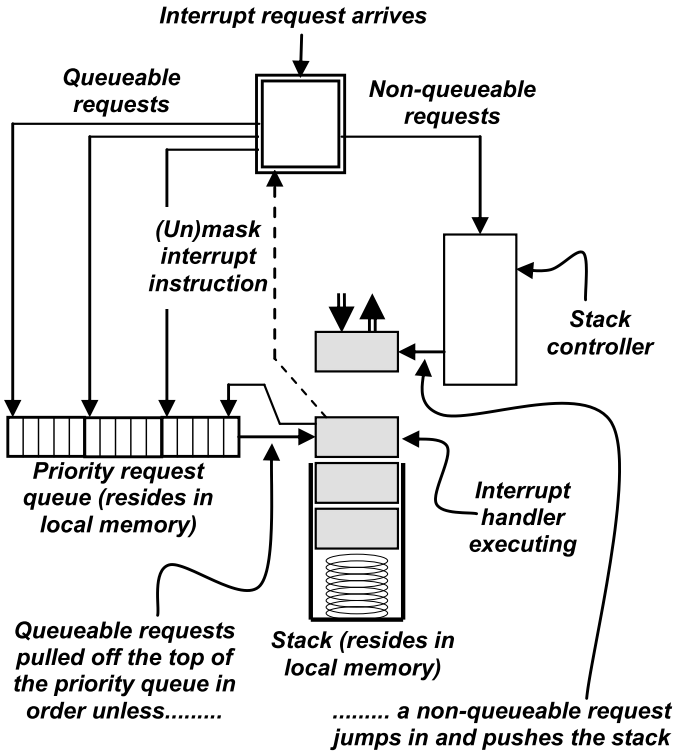


Figure 3.23: Per-core interrupt handling process [6].

All neuron models on the core are updated whenever the timer tick fires, but individual synapse models are updated upon packet arrival (i.e., *spike* arrival) [129]. Listing 3.1 provides an example of a packet arrival event handler. Synapses may connect to multiple neurons in the receiving core which is the purpose of the loop in lines 12–22. Typically synapse models prepare data for the neuron model to process when the next timer tick fires. While this system has been designed around spiking networks, it can also be applied to non-spiking networks with appropriate synapse models [130].

The leaky integrate-and-fire (LIF) [131] and Izhikevich [132] models are often used within SpiNNaker because they are computationally efficient whilst still retaining sufficient biological accuracy. Both neuron model coefficients and synapse weights are specified as parameters stored in the DTCM and SDRAM respectively. These parameters are run-time mutable which allows models to exhibit plasticity [82]. A numerical integration step is required to solve the differential equations, and as SpiNNaker does not feature

```
1  void updateActivationLevels() {
2      receiveSpike(p); //received packet p with source address
3      SDRAMB_lockBaseAddres = lookupTable.read(p.SourceID);
4      //DMA operation to read a block of memory
5      //from SDRAM to DTCM memory
6      requestDMAOperation(
7          SDRAMB_lockBaseAddres, &LocalSynapseBlock,
8          SynapticBlockSize, READ);
9      wait(DMACompletionInterrupt);
10     //For all neurons, update projected activation
11     //level as per the synaptic and delay information
12     for (int i=0; i<SynapticBlockSize; i++) {
13         //the real-time delay
14         int delay       = LocalSynapseBlock[i].Delay;
15         //neuron index
16         int neuronIndex = LocalSynapseBlock[i].NeuronIndex;
17         //synaptic weight
18         int weight      = LocalSynapseBlock[i].SynapseWeight;
19
20         neuronsState[NeuronIndex]
21             .ActivationLevel[currentTime + delay] += weight;
22     }
23 }
```

Listing 3.1: Pseudo-code for a neuron update [129].

any cores with a floating-point unit (FPU), fixed-point arithmetic is used throughout without significant loss of biological fidelity [133].

3.3.3 Abstract Time vs. Real-Time

At the core of a conventional discrete simulator is a queue of *events* which is used to advance the simulation. The head of this *event queue* is dequeued (*popped*) by a process called the *event pump*, implemented as a loop that terminates when the event queue becomes empty [134]. Dequeuing and dispatching an event to the parts of the simulator responsible for implementing the desired behaviour may generate more events, thus refilling the event queue. Eventually, the simulated behaviour should ideally converge and no new events will be generated. Clearly behaviours that fail to converge will cause infinite simulation times, and should therefore either be prevented or, at least, detected.

Events have a *time* associated with them when pushed into the queue. The queue is sorted by time to ensure that temporal causality is maintained; i.e., an event may never cause a new event to happen in the past or zero time in the future because that cannot happen in physical reality. Potentially, allowing events to have effects in zero time can lead to irrecoverable oscillations which prevent the simulation from ever finishing. However, certain simulation domains relax this constraint by merely limiting

the category of events that can be queued with zero time delay. Digital hardware simulators, for example, permit combinatorial logic events to be scheduled in this manner to allow equations containing feedback terms to be resolved. Clearly this can trivially lead to oscillations, hence these simulators place a limit of the number of event pump cycles that can take place in a single time-step and terminate with error if it is exceeded.

When the event pump pops an event from the queue, the timestamp is checked against the current value of simulated time (or *abstract time*). If they are equal, the event is processed as normal. If the new timestamp is greater than the current, simulation time is advanced to that point before the event is processed [135].

SpiNNaker performs simulations in real-time. Time therefore "models itself" because simulated time *is* wall-clock time. In a biological system, neurons take time to produce spikes, axons take time to transmit them, and synapses take time to accept them. By contrast SpiNNaker is infinitely fast; MC packets (spikes) propagate at a rate of a few hundred nanoseconds per inter-node hop and a neuron can be updated in a few microseconds, whereas biological systems tend to operate on a scale of 10's of milliseconds. Essentially, this means that new events can arrive in zero time, thus potentially violating causality.

SpiNNaker overcomes this potential difficulty by using a regular timer tick to schedule updates to the numerical parts of the code. The processing and delivery of spikes may happen in "zero time," but new events are not (potentially) created until the next firing of the timer tick. Axon delays are modelled in a similar way: all synapses have associated with them a delay parameter which is a multiple of the timer tick (usually about 1ms), and are only delivered (locally) when this delay is met. For example, if a spike arrives at a processor and a delay of 3ms is expected to model the axon delay, it will be placed in a queue of pending spikes with its delay noted against it. On each timer tick, every spike in this queue has its outstanding delay decremented by the timer interval, and are delivered once this reaches zero.

Axon delays are therefore modelled at the destination rather than at the source. This also highlights another difference between SpiNNaker and a conventional simulator: there is no global event queue—SpiNNaker offers parallelism *at the event level*. Each processor handles events that relate to the neurons it hosts locally. Clearly maintaining global synchrony on a machine of this scale is not sensibly achievable, but the skew between timer ticks is not sufficient to affect biological simulations [136].

Finally, it is worth noting that the timer tick must be set so that the numerical stability of the equations being integrated is maintained [6]. Provided that the timer tick is fast enough to ensure this is true and but also slow enough to give the processors sufficient time to be *infinitely fast* with respect to biology, simulations can function correctly without running into oscillatory issues caused by zero-delay events or numerical instability of the integration scheme.

These assumptions have allowed SpiNNaker to successfully and accurately simulate systems of LIF neurons [131] and multi-layer perceptron networks [130].

3.4 Non-neural Simulations

SpiNNaker makes several assumptions to ensure that the biological simulations are stable and accurate. To apply this machine to a broader range of problems requires that they are reformulated within these constraints. In the most general case, an application on SpiNNaker is a graph where the nodes represent a fragment of functionality and the edges identify a communication path. These may have additional processing associated with them if required (as is the case in neural simulation with the synapse models).

Any problem that can be discretised in the manner described here can be formed into a SpiNNaker application. Consider the case of a digital circuit simulation: the *problem graph* is clearly the circuit itself, and the *problem devices* (i.e., graph nodes) are the digital components of the circuit. Every computation that results in a state change causes an MC packet to be sent to all appropriate circuit nodes, containing the new state and a suitable timestamp in the future representing the propagation delay of the gate. Maintaining causality without a central event pump (as is the case in SpiNNaker) requires careful algorithmic design [137], which is beyond the scope of this thesis (see Bai [138]).

Consider also any situation that involves large matrix and vector operations such as image processing. Each matrix cell is be a problem device, and the problem graph describes the association each cell has with its neighbours. Many matrix operations then become $O(n)$ because the sum and multiplications for each new cell can be performed in parallel. Massive arrays of pixels can be streamed into applications structured this way to achieve real-time video processing. Here, the timer tick serves the same purpose as with neural simulation, both to maintain causality and the numerical stability of the algorithms.

Spatial problems can be discretised in such a way that a problem device represents a fragment of the space, and updates according to the space-fragments around it. This idea serves as the basis for Chapter 6 where an application is introduced to solve heat diffusion equations. The problem itself is not novel, but the structure of the application demonstrates that non-neural problems mapped into the biologically-imposed constraints of SpiNNaker produce correct results and scale well. Parts of this work have been published as Brown, Mills, Dugan, *et al.* [7].

Chapter 4

System Configuration

The first goal of this work (outlined in section 1.4) is to produce a set of algorithms that can accurately survey a target SpiNNaker system to determine any and all faults that may have occurred. Recall the tool-flow shown in Figure 3.19. Here both the topology and a fault-map of the target machine must be known *a priori*. Without means to reliably detect these faults, other than manual inspection, these files are usually a best estimate based on what the structure of the machine *should* be.

Figure 4.1 shows the same tool-flow but modified for the purposes of the work in this thesis. There are no longer any hardware maps that must be known ahead of time. Instead the machine topology and the fault map are discovered from the live machine being targeted.

Referring to Figure 4.1:

Loader is introduced in the previous chapter as the tool that takes an application description (a *problem graph*) and produces a set of data structures and MC routing table entries to be written into the machine.

Uploader is introduced in the previous chapter as the tool responsible for writing these data-structures, along with compiled binaries, into the machine. Uploader also produces a set of 'C' header files that user code can use to reference named parts of the problem graph. It is also capable of dynamically adjusting the routing to bypass cores that may have failed after the POST.

Cross-compiler is a commercially-available ARM compiler [139] used to target the SpiNNaker processors. In this case, the `armcc` compiler from the RealView Development Suite v4.0 is used.

IntHand is the Interrupt Handler support library which responds to packet and timer interrupts and converts them into problem device, edge, and simulation update events for applications to handle.

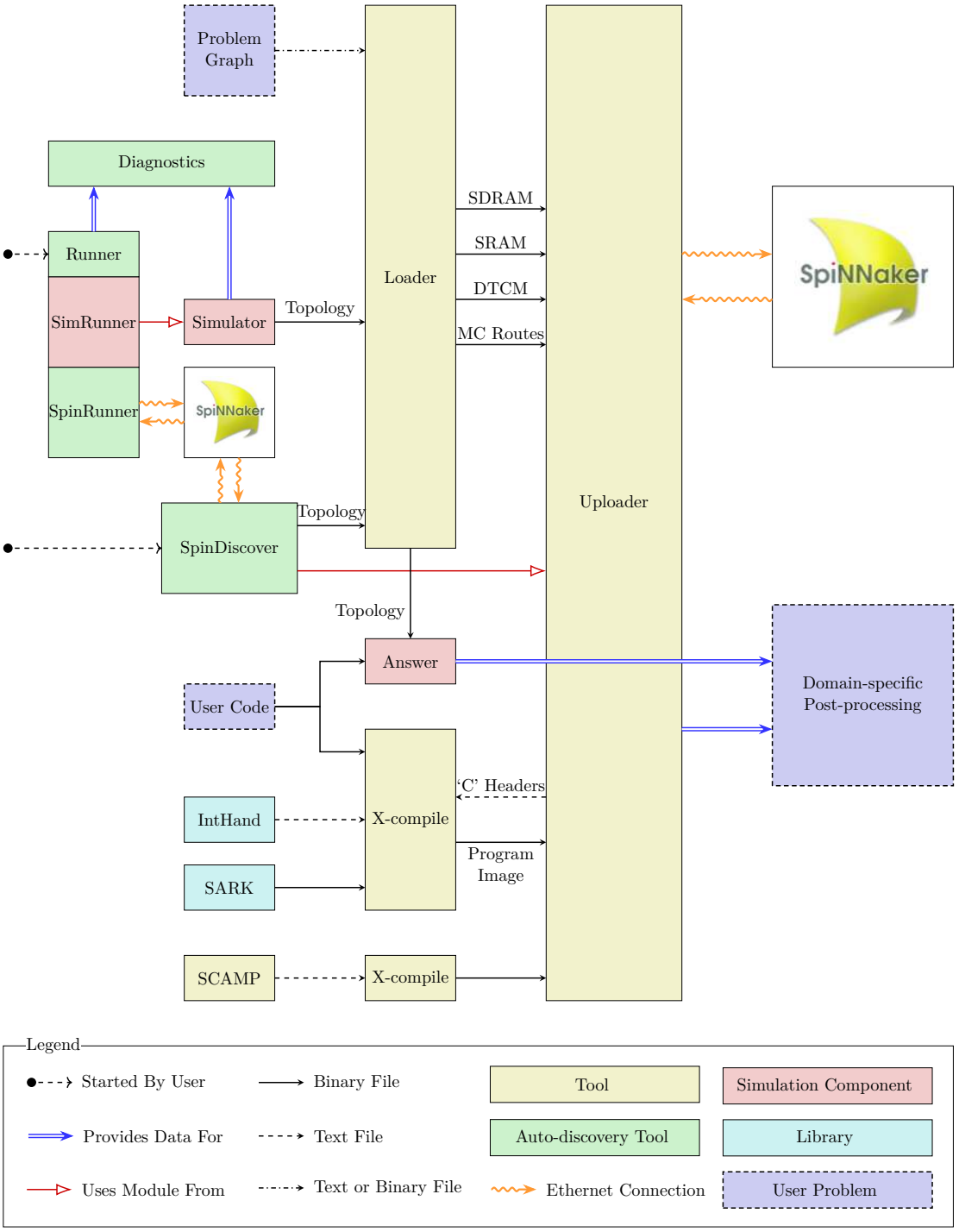


Figure 4.1: Complete revised SpiNNaker tool-flow factoring the contributions of this thesis.

SARK is the SpiNNaker Application Run-time Kernel introduced in the previous chapter which provides the event pump, an API to access the SpiNNaker hardware, and provides a harness for applications to fit into.

User Code defines the behaviours of the individual components of the application. For neural simulations, these are the neuron and synapse models. For non-neural

applications, these are the problem device and edge behaviours. User code must be linked against both SARK and IntHand to form a valid SpiNNaker application.

Answer is a conventional single-threaded discrete stand-alone simulator that can execute user code in a SpiNNaker-like environment, allowing behaviours and models to be verified before being uploaded to SpiNNaker.

SpinDiscover serves as the host-machine interface to the topological discovery algorithms presented in Chapter 5. It uses the API provided by the Uploader to send and receive special-purpose SCP commands to start discovery processes and receive the resulting data. From this, a *binary topology file* is produced that accurately describes the current hardware state of SpiNNaker (including faults) that the Loader can use to map applications correctly.

SpinRunner is a tool for measuring the performance of these algorithms on the SpiNNaker hardware. It uses an API provided by SpinDiscover to achieve this, and is capable of generating charts showing the performance results as well as graphics describing the discovered machine state.

Simulator is a module for SpinRunner that simulates just the communications of a real SpiNNaker machine. This tool is used to construct models of the algorithms to ensure the behaviours were correct, before the SpiNNaker implementations are created.

SimRunner provides a front-end for SpinRunner when the Simulator is used as the back-end instead of SpiNNaker. It shares the core of SpinRunner to maximise code-reuse.

SCAMP is introduced in the previous chapter as the SpiNNaker Control and Monitoring Program and provides the external command interface for controlling a SpiNNaker platform. Essentially this is the "system software" for SpiNNaker.

4.1 User Applications

4.1.1 Definition

Applications are described in two parts: firstly, the problem graph is a textual representation of the connectivity of the problem devices of a problem. An example problem graph is shown in Figure 4.2.

`D1`–`D8` are *problem devices* that perform some computation on the data that they are given, resulting in more data flowing to the other problem devices (or perhaps back onto themselves if that makes sense in the context of the problem at hand). The *edges* represent a unidirectional flow of data from one problem device to another, optionally

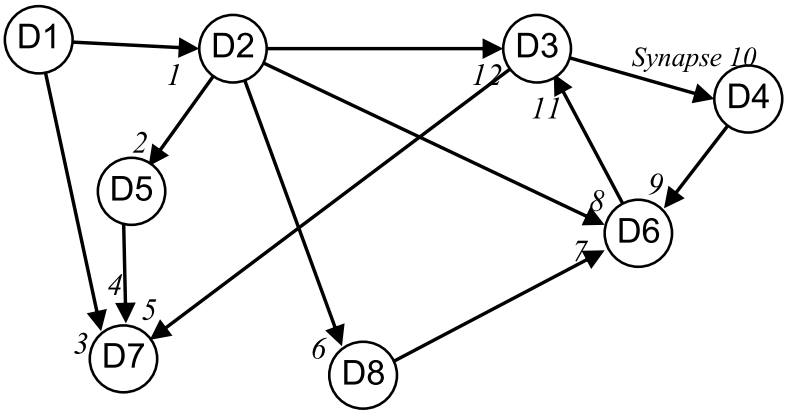


Figure 4.2: Example problem graph [6].

with *pins* at end serving as a generalisation of a synapse. Fan-ins and -outs can range from just a single connection up to about 10^5, and are limited by the amount of available SDRAM space.

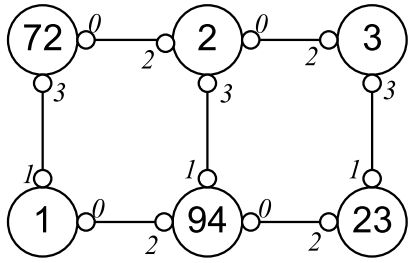
Each problem device and edge can have associated *parameters* and a memory requirement for *state*. These are both assembled into data structures that will ultimately be written into the node-local SDRAM and SRAM, and the core-local DTCMs. Parameters are nominally immutable and state is nominally mutable with definable initial values, but this is essentially just a convention as no attempts to enforce this are made at run-time for performance reasons.

The second part of an application is the set of *handlers* that implement the behaviour of the problem devices and edges. These are written as conventional 'C' functions which are registered as callbacks in SARK and IntHand. SARK implements the timer tick handler and the packet arrival handler, and then IntHand automatically registers itself against these to provide specific callbacks for individual device updates, edge updates following a packet arrival, and a *simulation step* update after everything else in the timer tick has been serviced. Typically, only the device and edge updates are used, but the simulation step update could be used to take a snapshot of the current state of a core for later inspection.

4.1.2 Compilation

Problem graphs must be mapped to a particular machine topology for the resulting data-structures to make any sense. Figure 4.3 shows an example simple machine configuration that is used in this section. Specifically, Figure 4.3a shows the hardware connectivity of the example machine, and Figure 4.3b shows the P2P table configuration for each node.

The Loader accepts both the problem graph and the machine topology as either ASCII text or binary files. A model of the machine topology is constructed in memory onto



(a) Example SpiNNaker machine topology.

Target Node \ Node	1	2	3	23	72	94
1	L	2	2	2	3	2
2	0	L	2	2	0	1
3	0	0	L	1	0	0
23	0	3	3	L	0	0
72	1	2	2	1	L	2
94	0	3	2	2	0	L

(b) P2P configuration for the example machine topology.

Figure 4.3: Example SpiNNaker machine configuration [6].

which the problem graph is mapped as shown in Figure 4.4. In many ways, this is similar to the place-and-route stage present in many FPGA design tools. Many of those algorithms are also applicable to this process.

Problem devices are assigned to physical SpiNNaker cores that have sufficient memory to hold the parameters and state. A user can constrain particular problem devices to be resident on specific cores if require. For example, perhaps the model is particularly complex and requires the compute resource of a dedicated core.

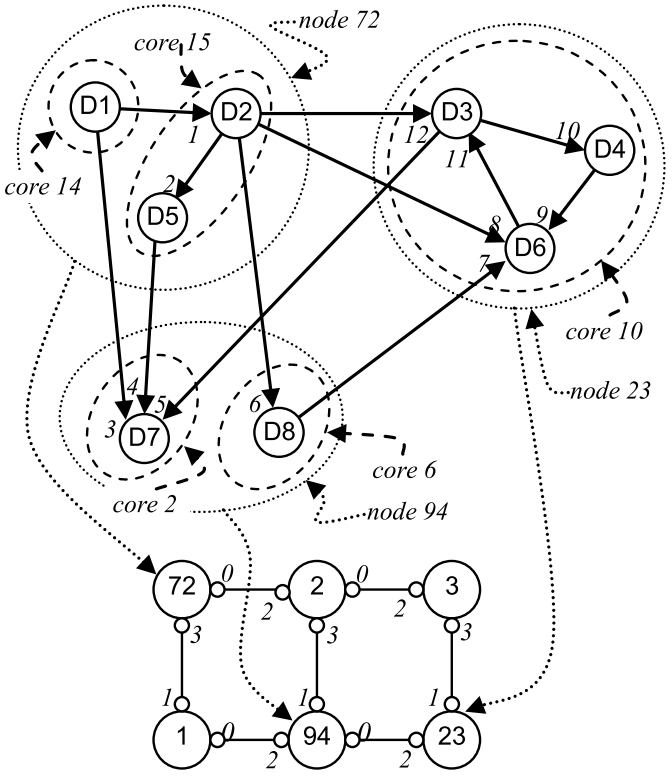


Figure 4.4: Mapping the example problem graph (Figure 4.2) to the example machine topology (Figure 4.3a) [6].

Once this mapping has been performed, the Loader then computes the MC routing tables as shown in Figure 4.5. MC routes typically follow P2P routes over long distances, which can be seen by the problem edges flowing through Node 2 despite it housing no problem devices itself. Each node has a Device Lookup Table (DLT) which maps inbound packets onto the correct problem edge/device. Recall that AER stores the *source* address in the packet and not the destination. The DLT provides a mechanism for the receiving core to perform the required cross-reference from the source to all receiving problem devices. IntHand automatically performs this translation for all applications that link against it.

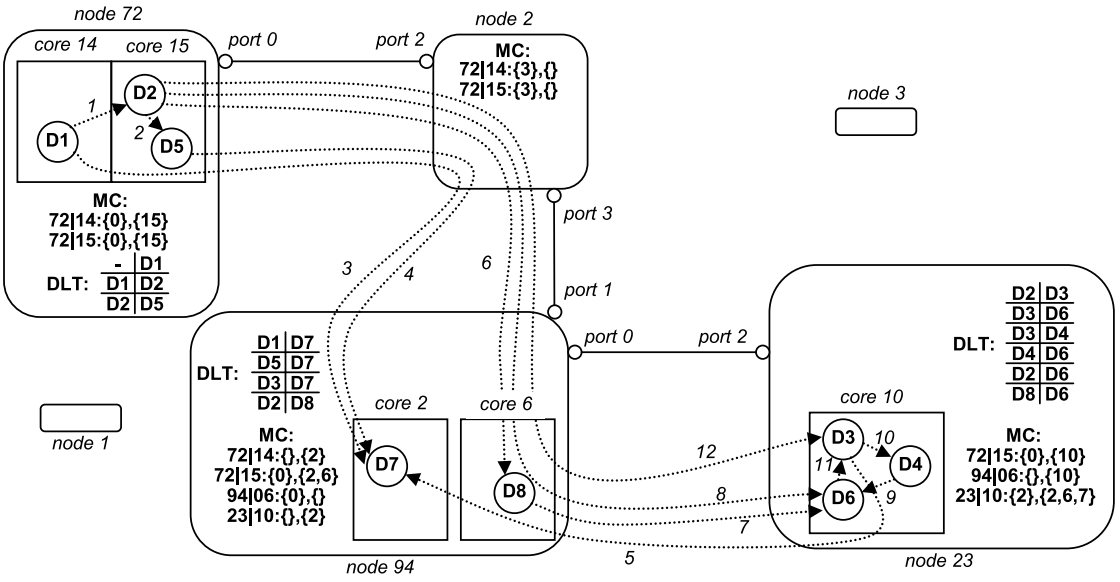


Figure 4.5: Data-structures for the mapping in Figure 4.4 [6].

This translation is non-trivial and is perhaps best illustrated by example. Consider the routing required for D1 in Figure 4.2. D1 requires edges to both D2 and D7 which are all mapped to different cores as shown in Figure 4.5. The simplest of these routes is the D1 → D2 edge which is completely resident in Node 72. Whenever Core 15 receives a message from problem device D1, it must search the DLT for all connections originating from D1 and deliver the message to *all* appropriate local problem devices. The DLT for Node 72 contains only a single mapping for D1, (D1: D2), and hence only device D2 will be issued with the message. To implement the D1 → D7 problem edge, the Node 72 router delivers messages to Port 0 which leads to Node 2. Node 2 forwards these out of its Port 3, causing them to arrive at Node 94, where its router will deliver the messages to Core 2. The DLT here also contains only a single mapping for D1, (D1: D7), and hence Core 2 will issue messages to only device D7.

It is acceptable for a DLT to include multiple mappings. Consider the routing where device D2 is connected to D3, D6, and D8. Both D3 and D6 are resident in Core 10 of Node 23, therefore requiring two entries in the DLT of Node 23: (D2: D3) and (D2: D6). When receiving messages from D2, Core 10 necessarily delivers them to both D3 and D6, thereby implementing the desired behaviour of the original problem specification.

The outputs from the Loader are a set of record-oriented binary files containing:

1. Edge/pin parameters and state destined for node SDRAMs (which are arranged as a tree) that forms part of the DLT.
2. A format mapping table for each SRAM which IntHand uses to unpack data correctly before issuing it to event handlers in the application.
3. A problem device state table for each DTCM to store state and parameters for the problem devices themselves. This forms the other part of the DLT for IntHand.
4. MC routing tables for each node, even if there are no problem devices mapped onto it (e.g., node 2 in Figure 4.4).

From these output files, the Uploader generates a pair of header files. One contains base addresses of the locations of the above data-structures so that IntHand knows from where to acquire the data it needs. The other defines constants based on the types of problem device the application uses. For example, if the problem graph defines a device type `NEU`, the header file would contain a constant, `TYPE_NEU`.

User code can now be compiled against these header files, IntHand, and SARK to produce a binary that can be uploaded. Typically the same binary is used in every core (i.e., SPMD), but this is just a matter of convenience; the tool-chain supports an MPMD model if this is more appropriate.

4.1.3 Execution

The Uploader begins by writing the node-local data into the SDRAM and SRAM of the various SpiNNaker nodes. Whilst this is happening, the header files (previous section) are generated to reduce the time it takes to upload an application. This is particularly useful for very large data-structures where performing multiple passes is too time-consuming. If the header files are produced in this step rather than the previous, invocation of the cross- compilers must be delayed until after the data has been uploaded. Typically, compiling the 'C' code takes an insignificant amount of time relative to the processing of the problem graph.

As mentioned in the previous chapter, the Uploader performs a "core liveness" survey while the node-local data is being written into memory. Single core failures are dynamically mapped around by the Uploader re-writing parts of the MC tables as they are being uploaded. DTCM data is written into the memories at the same time and is also subject to any remaps. However, multiple core failures must be known *a priori* so they can be mapped around by Loader.

Application binaries are then flood-filled (SPMD) or directly loaded (MPMD) into the instruction tightly-coupled memories (ITCM) of the appropriate cores, again subject to any remaps. Once the entire machine has been prepared with data, routing, and application binary images, a start SCP command is flood-propagated across the machine and the application begins executing.

4.2 Contributions

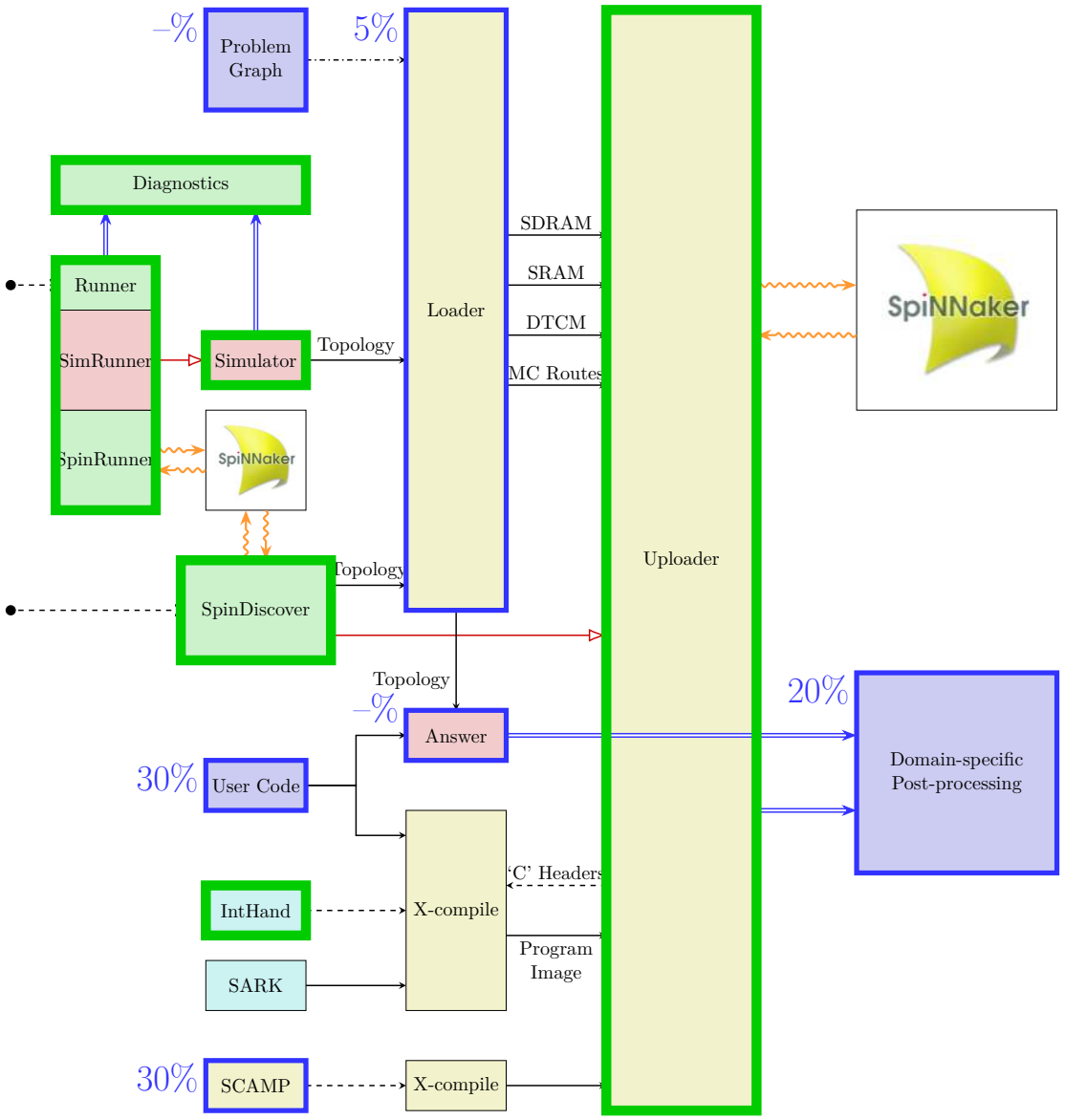


Figure 4.6: Tools and stages of the tool-flow that have been directly created or modified by the work in this thesis.

Figure 4.6 shows a shaded variant of Figure 4.1 which highlights the contributions of this thesis:

- Uploader and IntHand [127] were written specifically to support SpiNNaker applications. Uploader additionally provides an API that many other tools use to communicate with SpiNNaker.
- SimRunner and the corresponding simulator were created to verify the behaviour of the algorithms in Chapter 5.
- SpinRunner allows the SpiNNaker implementations of these algorithms to be appropriately benchmarked and tested.
- SpinDiscover wraps this functionality into a tool that can inter-operate with Loader.

In addition to the above, the following aspects were achieved in collaboration with colleagues, shown by the blue boxes and percentage contribution estimates in Figure 4.6:

- SCAMP was initially written entirely by researchers at the University of Manchester to serve as the system software. The algorithms of Chapter 5 target the system-level of SpiNNaker, hence SCAMP is the most logical vehicle to use to test the implementation of these algorithms. A benefit of this approach is that the work of this thesis is already integrated and can be used by any users of SpiNNaker with only minor modification.
- The heat diffusion application described in Chapter 6 is a collaborative effort. This work contributed (in addition to IntHand) a worker library that served as a high-performance buffer for inbound packet traffic to improve the run-time of the application.
- General-purpose data download and collation tools were produced to aid with the heat diffusion application, but are also transferable to other applications. A specialised system, SpiNNterceptor, is described in reference [123] which generalises data download to include plug'n'play peripherals that can be accessed from within SpiNNaker applications.
- Finally, various technical discussions and small contributions were made to the Loader, Answer, and the problem description for the heat diffusion application.

Chapter 5

Auto-discovery Algorithms

5.1 Problem Statement

Unlike more traditional parallel computers, SpiNNaker cores do not have a record of any other cores in the system, meaning the entire structure of the machine is an unknown. Boot-time discovery of the machine is essential to initialise the P2P routing tables so that arbitrary core-to-core communication can be achieved. As a consequence, core-level synchronisation is difficult to achieve, as is completion detection and the notion of *pausing* an application after it has started. In the neural domain, these issues are unimportant because the simulations are performed in real-time so the duration for which a simulation executes is itself a parameter of the experiment.

SpiNNaker is essentially an enormous number of ARM processors that can be applied to problems outside of the neural domain. However, this requires that certain biologically-inspired assumptions are slightly altered to be more general.

Firstly, any hardware faults in the machine must be known so that applications can be accurately mapped onto *specific* machines. It is currently assumed (by Loader) that all nodes have 16 worker cores, but the existing constraints capability allows any number of cores per node to the specified. The physical hardware has 17 worker cores, leaving one as a *spare* that can be brought into service automatically by Uploader if a single core fault is detected. It is also assumed that all ports of a node are always functional, and that if a port were to fail, this would be entirely mitigated by the toroidal network and emergency routing. These assumptions have not prevented any applications from being mapped to (and subsequently running successfully on) a specific SpiNNaker machine. However, it would be naive to assume that a machine of this scale will not develop faults as the components age.

Solving this problem is straightforward: a survey of the system before any problem-mapping (i.e., the Loader run on Figure 3.19) can determine any port failures and

detect exactly how many usable workers each node has. The latter is already recorded in each node as a consequence of the master arbitration step in the boot. The former is more difficult because it requires a de-centralised survey (recall that at this point, P2P communications might not be available).

Secondly, for the mapping stage to be successful in light of this newly discovered information, it must be possible to assemble this information into a data-structure that describes the *exact* machine topology. It must not be possible to over-allocate problem devices to a node just because it is assumed that a core *should* be there. This is shown as the "topology" arrow between the *SpinDiscover* and *Loader* blocks in Figure 4.1.

This issue is compounded by the non-deterministic nature of the P2P building process (section 3.2.1). P2P discovery messages are sent out of the northern, north-eastern, and eastern ports to flood over the machine, and wrap when the machine limits (which must be known *a priori*) are reached. Each message arrival is therefore a race condition, making the exact composition of the P2P tables non-deterministic. Loader makes use of the P2P tables to determine multi-hop MC routes. An exact mapping between an idealised model and a fault-free system is required to yield optimal MC routes, and a port failure automatically causes a discrepancy that could lead to application binaries that will not function on SpiNNaker. As mapping can take many hours on a conventional for large problems, this is clearly unacceptable. Packaging complete and accurate P2P tables into the topological information allows this to be avoided entirely.

Finally, embedding a *control tree* in the system establishes a parent/child hierarchy that can be used to provide familiar synchronisation and completion-detection functionality. Perhaps the most obvious function this can serve is a *barrier* implementation, described in Chapter 2, where cores enter the barrier once they have finished their assigned task and all their child cores have also entered the barrier. Once the root core is in the barrier state, clearly so is the entire machine [140]. The control tree can also be used to implement other parallel programming concepts that a particular application may require.

In summary, broadening the scope of SpiNNaker to include select non-neural applications requires that:

1. Any hardware faults are discovered before any mapping takes place,
2. A complete model of the current state of the hardware can be assembled outside of SpiNNaker based on data obtained from any discovery algorithms, and
3. A control tree can be embedded in any SpiNNaker machine to support common parallel programming constructs.

This chapter provides solutions to these problems that have been verified both in simulation and on SpiNNaker hardware.

5.2 Definitions

All of the algorithms in this chapter use the following notation and are implemented under the following assumptions:

- The algorithms are system-level and hence execute on the monitor core (see section 3.2) of a node. Any packet traffic arriving at a node is forwarded directly to the monitor to progress the algorithm.
- Figure 3.20 shows a SpiNNaker network fragment, where each node has six discrete ports represented by circles containing compass directions, and the local *virtual port*, L, that leads to the monitor. Ports are bidirectional but faults may cause them to function in only a single direction or not at all.
- Algorithm implementations (both in simulation and on SpiNNaker) use a numerical scheme to index ports instead of the compass directions. This begins at 0 for the eastern port and continues anti-clockwise; i.e., 1 for north-eastern, 2 for north, etc.
- Each algorithm is implemented as a state-machine that each node is running *simultaneously*. States are shown *in this style*.
- Nodes communicate using *tokens* which are written **in this style**. A token may optionally carry parameters between nodes and are shown as **token**(A, B, C), where A, B, and C are the parameters.
- Ports also have states shown in this style.
- All nodes have a unique identifier as explained in Chapter 3; ports may also have a node identifier assigned against them to note which node they connect to.

5.3 Simulation

SpiNNaker does not have an interactive debugging capability, which makes designing entirely new applications difficult. In the neural domain, applications are assembled from individually tested models by automated tools. Most of the uniqueness of an application is in the connectivity of these models rather than the code itself.

Figure 5.1 shows the part of the tool-flow designed in this section. The focus of the Simulator block is to allow system-level algorithms to be designed in a conventional environment before being moved to SpiNNaker. Conventional debugging and testing approaches are used to check that the algorithm performs the required behaviour, thus minimising the scope for error when creating the SpiNNaker implementation.

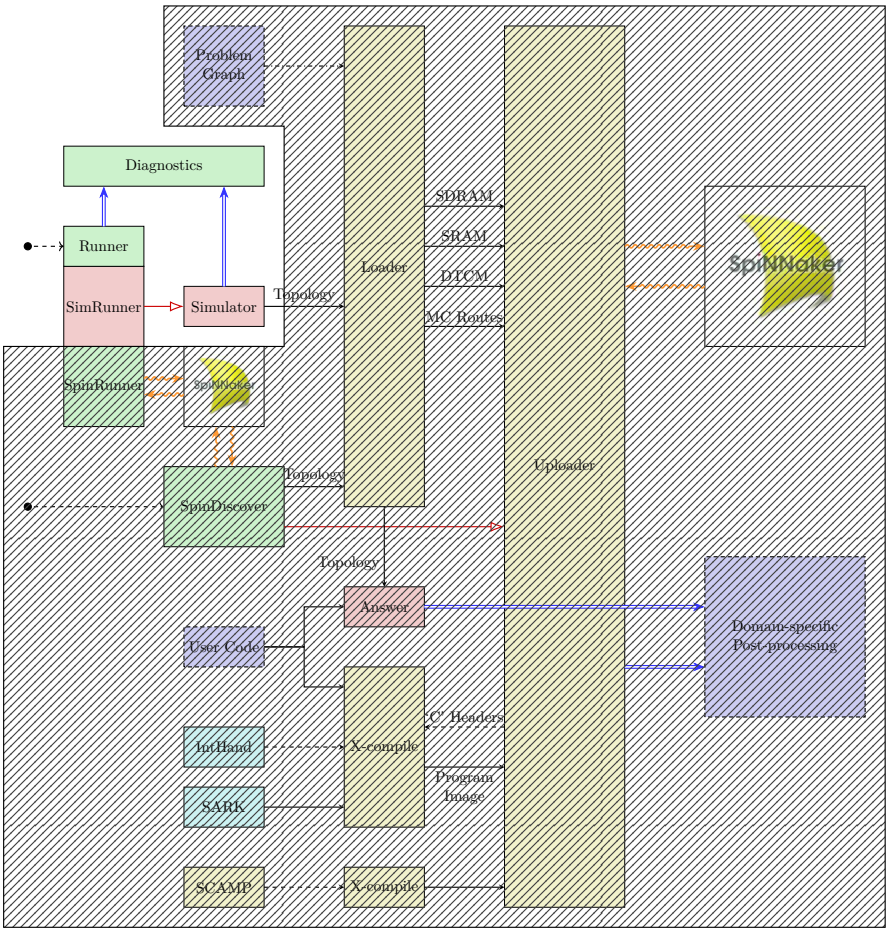


Figure 5.1: Stages of the toolflow shown in Figure 4.1 required to design and simulate the behaviour of the auto-discovery algorithms.

As Figure 5.1 shows, the Simulator block is used by SimRunner which is itself a front-end for Runner. The system has been designed to present the user with roughly the same command-line interface regardless of which back-end is used. However, the Simulator can be parametrised in more ways than SpiNNaker and has additional instrumentation that is either not possible and/or not applicable to the SpiNNaker implementation.

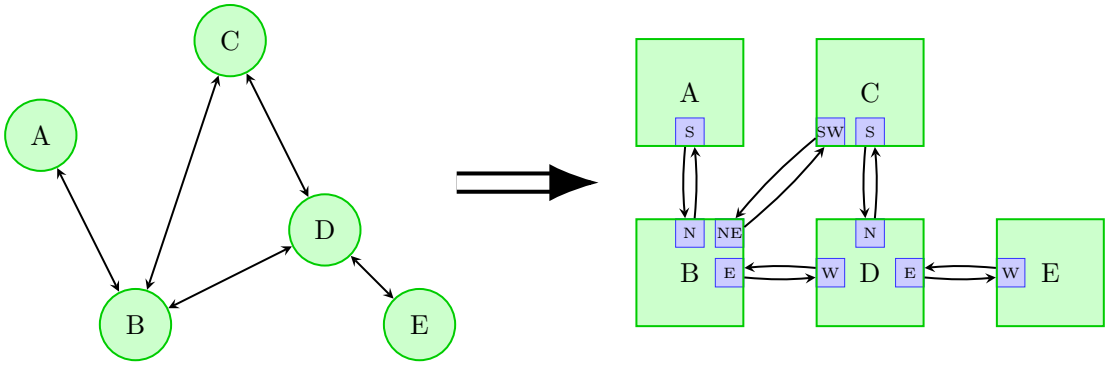
Runner encapsulates the shared experimentation framework that measures the run-time of a particular algorithm and produces appropriate graphs of results. The simulator (and SimRunner) are used to test and validate the rest of the Runner framework and verify that the simulator can perform workload-realistic simulations.

5.3.1 Machine Model

For the simulation to make accurate predictions, it must hold a model of a physical SpiNNaker machine internally. The simulator is written in Python 2.7 and uses object-oriented programming (OOP) techniques where appropriate. Each SpiNNaker node is represented by a `MachineNode` object. Recall from earlier that all algorithms are

addressing *system-level* issues that arise during the boot procedure. Therefore only the monitor core is required and only NN communication is available.

Connectivity is critically important in the model just as in a physical SpiNNaker machine. Each `MachineNode` holds a set of `MachinePort` objects which each contain a reference to their parent node and the target node, thus providing unidirectional communication to immediate neighbours. Port objects have a *physical* and a *virtual* state which signify their availability to the algorithms. Physical state refers to the connectivity of the port, whereas virtual state is used to apply constraints to the algorithms in some way. For example, a port could be physically connected but virtually disabled, which would be seen as disabled from the perspective of any algorithm. Port objects also have arbitrary parameters that are used to hold algorithm-specific information.



Abstract Connectivity Graph Machine Graph

Figure 5.2: Constructing a machine model from an abstract connectivity graph.

`MachineNodes` and `MachinePorts` form a *machine graph* which is assembled from an input *abstract connectivity graph* as shown in Figure 5.2. This mapping requires care because the compass direction of a port is determined from its index (and *vice versa*). This is achieved by forcing all nodes in the abstract connectivity graph to use (X, Y) co-ordinate pairs as their unique identifier. A stencil function is applied to all nodes of the graph to determine the set of 'SpiNNaker neighbours' for the node, which directly leads to the `MachinePort` objects being instantiated with the correct source and target nodes.

Square and rectangular lattice topologies can be generated in the simulator with the appropriate SpiNNaker connectivity pattern if required. Alternatively, a topology file, specified using the widely used GraphML[1] format, can be used. Figure 5.3 shows a visualisation of the topology file that describes an ideal SpiNN-103 (Figure 3.15) system, which is used for all simulations in this chapter unless stated otherwise.

[1] http://graphml.graphdrawing.org/

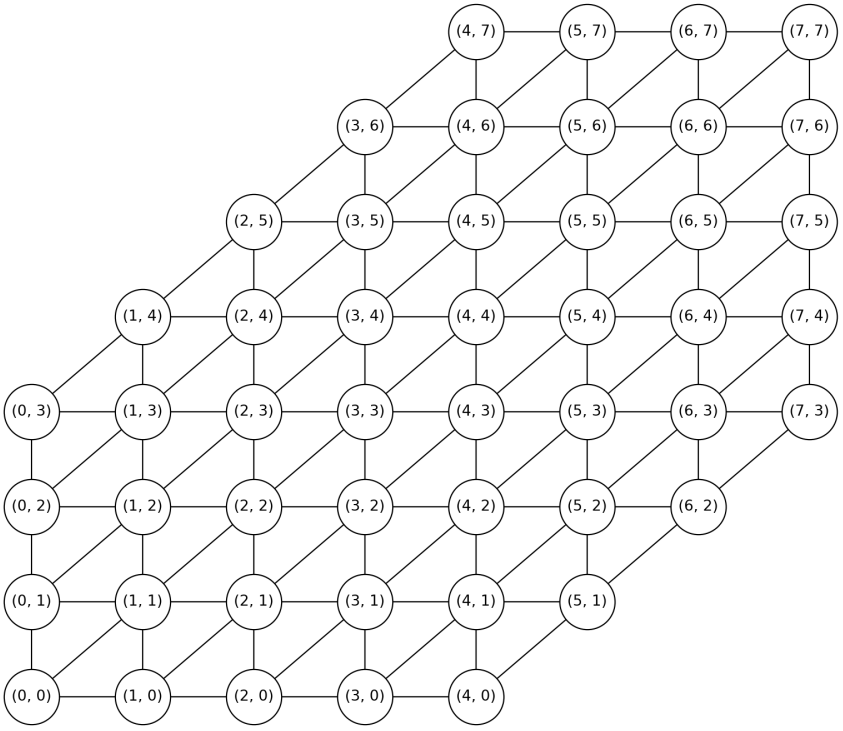


Figure 5.3: Abstract graph representing the connectivity of a SpiNN-103 system (Figure 3.15), showing the effect of the connectivity pattern of Figure 3.17.

5.3.2 Simulator Architecture

Each `MachineNode` can hold one or more `Behaviour` objects which implement a specific piece of the algorithm under test. Packets that arrive at a node are passed to any active `Behaviours`, which in turn delegate to a specific handler function in a similar manner to SpiNNaker. Multiple `Behaviours` can be active simultaneously, but typically only one is required.

Inter-node communications are handled by the `MessageBroker` object shown in Figure 5.4. It is responsible for queueing simulation events with the correct timing to simulate a combined transmission and computation delay. All `MachinePort` objects hold a reference to the `MessageBroker` to ensure the correct timing is always observed. If this were not the case, events can be created for the current time-step which, as mentioned in Chapter 3, can cause oscillations as causality has been violated.

The `MessageBroker` converts messages into time-stamped events which the event pump uses to progress the simulation. Events may have arbitrarily many parameters associated with them, and are divided into *categories* that can each have a number of *receivers* associated with them. Whenever the event pump pops an event from the queue (which is ordered by event timestamp), the event and all parameters are sent to *all* handlers registered against the event category. In most cases in this chapter, the `MessageBroker` is the only receiver and `comms_event` is the only event category used. Figure 5.4 shows

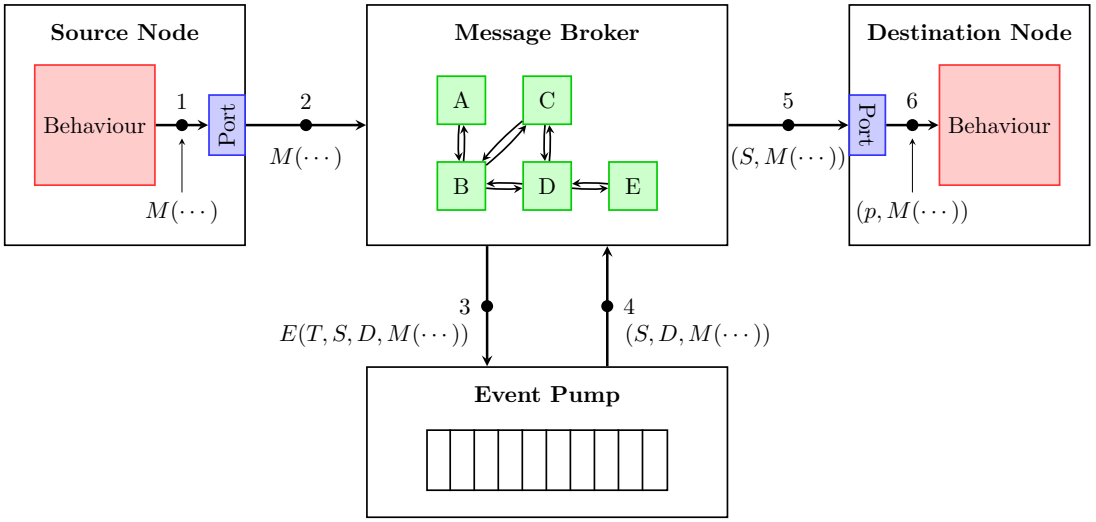


Figure 5.4: Message delivery system in the simulator.

this architecture, identifying key points of a node-to-node message transmission, detailed as follows:

1. A `Behaviour` in the source node emits a message, $M(\cdots)$, with some parameters through the appropriate port, which also determines the name of the destination node in the topology.

2. The `MachinePort` passes $M(\cdots)$ to the message broker, along with the source and desired destination node names.

3. An event, $E(T, S, D, M(\cdots))$, is issued to the event pump where T is the timestamp of the message delivery event, and S and D are the source and destination node addresses. T is calculated by the `MessageBroker` adding a suitable delay onto the current simulated time to model the lumped transmission and computation delay.

4. Once simulated time has advanced sufficiently to allow E to be popped from the queue, the event pump issues the message broker (a *receiver*) with the tuple $(S, D, M(\cdots))$; i.e., the arguments stored against the event.

5. The message broker looks up the destination node, D, in the machine model and issues it with a tuple of $(S, M(\cdots))$.

6. S is used to find the port object, p, in the destination node that can be used to reply to S. The behaviour is then issued with p and $M(\cdots)$, potentially leading to state transitions and further messages being generated.

5.3.3 Fault Map Generation

To determine how the algorithm under test scales with the number of discoverable nodes, the simulator generates a *fault-map* before each run which selectively disables `MachinePorts` to lock certain nodes out of the simulation. Figure 5.5 shows a visualisation, produced by the simulator, of the machine graph generated from the topology shown in Figure 5.3. Each node is shown by a large square containing six ports shown as smaller colour-coded squares. In this diagram, red ports have malfunctioned (consider that there are no corresponding edges in Figure 5.3), and green ports are fully functional.

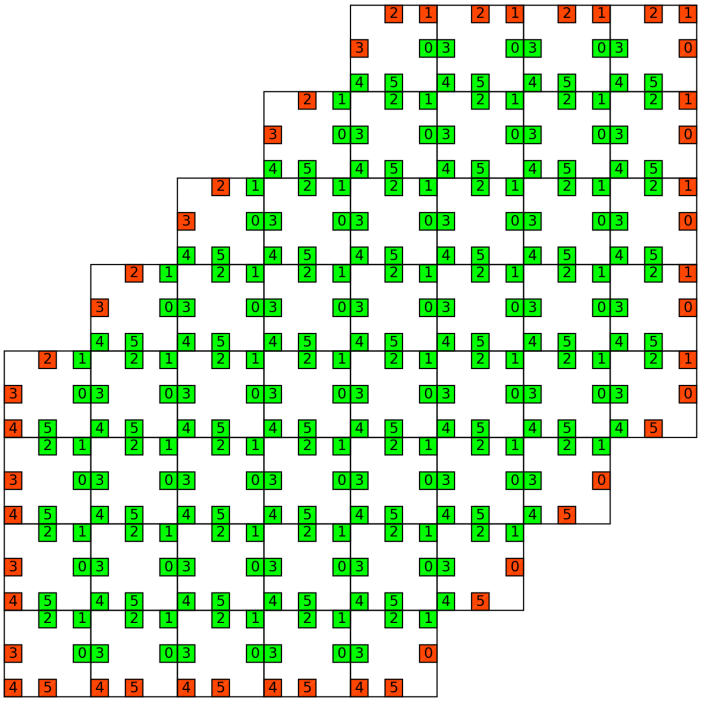


Figure 5.5: Visualisation produced by the Runner infrastructure of the machine model resulting from the graph shown in Figure 5.3.

Figure 5.5 shows only the *physical* state of the ports, but altering the *virtual* state allows the machine topology to appear smaller. The simulator generates fault maps by first setting the virtual state of all ports to disabled, and then N nodes are enabled row-by-row from the lower-left (root) node. This technique is used to measure how algorithms scale both in simulation and on SpiNNaker.

5.3.4 Test Case

To verify that the blocks highlighted in Figure 5.1 correctly interoperate, the simple case of the default P2P configuration algorithm detailed in section 3.2.1 is used. Only the node identifier assignment stage is implemented because, as discussed in section 5.3.1,

the `MachineNode` object does not contain a P2P router model (though adding this would require only minor modification). The outcome of this procedure is already known: all nodes will be assigned (X, Y) identifiers starting from $(0, 0)$—the *root*—in the lower-left corner of the model (Figure 5.5), and increasing in both X and Y up to the upper-right corner.

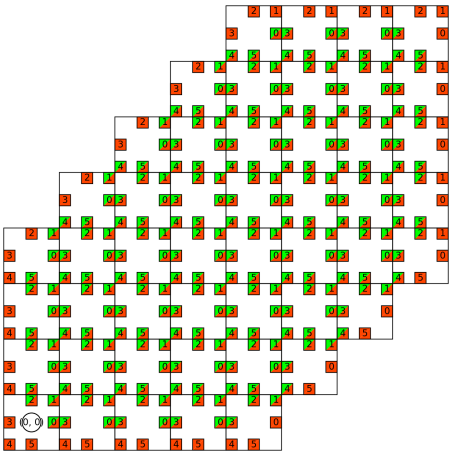
Algorithm 5.1 shows the algorithm implemented in each node, where W and H are the dimensions of the target machine which must be known *a priori*. These values are determined as if the topology traced a rectangular lattice that may be incomplete; e.g., Figure 5.3 is an incomplete 8×8 square lattice, therefore $W = H = 8$.

Algorithm 5.1 (X, Y) node identifier assignment algorithm.

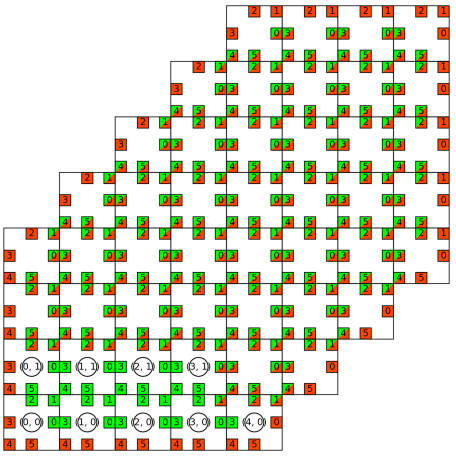
Require: W and H to be extents in X and Y known *a priori*.

```
1:  procedure nnHandler(p_s, X, Y, W, H)
2:      Δx ← Δy ← 0
3:      if p_s = W then
4:          Δx ← 1
5:      else if p_s = SW then
6:          Δx ← 1
7:          Δy ← 1
8:      else if p_s = S then
9:          Δy ← 1
10:     end if
11:     X' ← (X + Δx) mod W
12:     Y' ← (Y + Δy) mod H
13:     node.identifier ← (X', Y')
14:     for p_t ∈ {E, NE, N} do
15:         sendNN(p_t, (X', Y', W, H))
16:     end for
17: end procedure
```

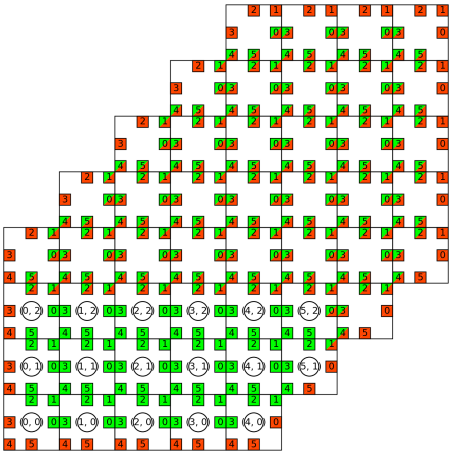
Figure 5.6 shows a range of machine model states after Algorithm 5.1 has finished executing. The trajectory of the fault-map described previously can clearly be seen progressing through the figures. As there are 48 nodes in the topology used to create the model, there are consequently 48 individual runs subject to 48 individual fault-maps. Port states are shown by the diagonal shading, in this case the green shading in the upper-left shows that the connectivity physically exists, and the red in the lower-right indicates that the port has been disabled. As expected, no node with all ports disabled has received an identifier because no messages have been received by the behaviour in the node.



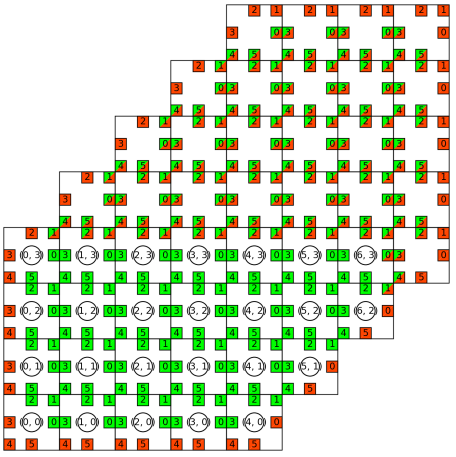
(a) Single node enabled.



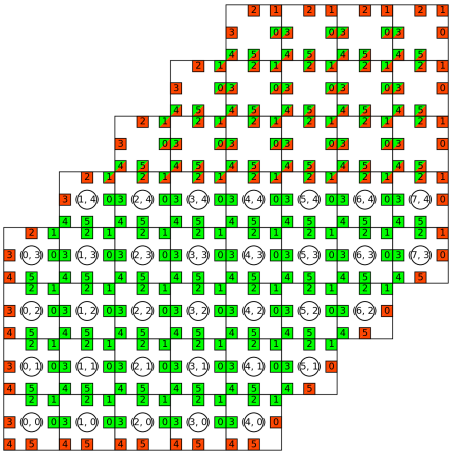
(b) 9 nodes enabled.



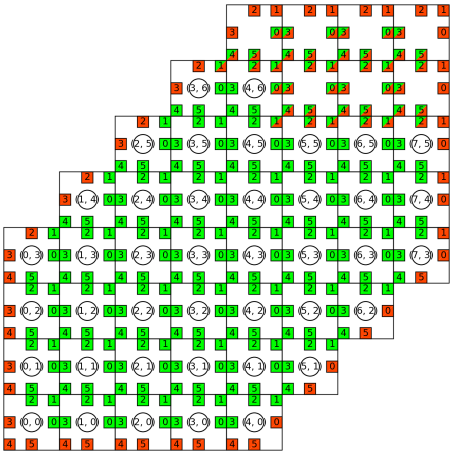
(c) 17 nodes enabled.



(d) 25 nodes enabled.



(e) 33 nodes enabled.



(f) 41 nodes enabled.

Figure 5.6: Machine model state after P2P algorithm has been executed on various fault maps limiting the number of discoverable nodes.

Figure 5.7 shows the final experiment run with a fault-map that enables all ports in the model. It is worth noting that the virtual state of a port can never upgrade the apparently capabilities of a port. Virtual states naturally degrade to the physical state if this does occur; for example, if a port is physically disconnected and the virtual state is set to enable it, all subsequent *reads* of the virtual state will still state that the port is disabled.

Algorithms are therefore always constrained to the safest possible case and will never expect responses from ports that can never produce responses. This is simple to achieve in the simulator because the topology is known. In SpiNNaker, the topology is clearly not known (as discovery is the purpose of the algorithms), so a port survey process called the ***α*-ping** is run first to determine the physical port states. This is described in the next section.

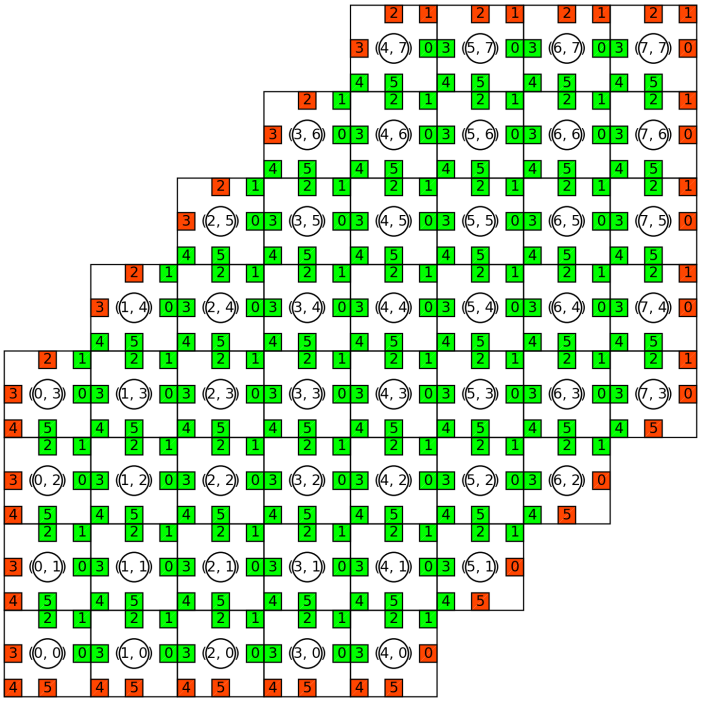


Figure 5.7: Machine model state with all 48 nodes enabled clearly showing the correct allocation of identifiers.

Figures 5.8 and 5.9 show two of the diagnostic plots generated specifically from the 48 node run (Figure 5.7). Figure 5.8 shows how the simulator advances simulated time as events are popped from the event queue. These simulations were performed on a desktop machine with a 3.0GHz Intel i7-950 processor and 24GB of RAM.

The widths of the bars of Figure 5.8 change with height to show how much extra processing time is required to dispatch and handle the events. From Algorithm 5.1 it is

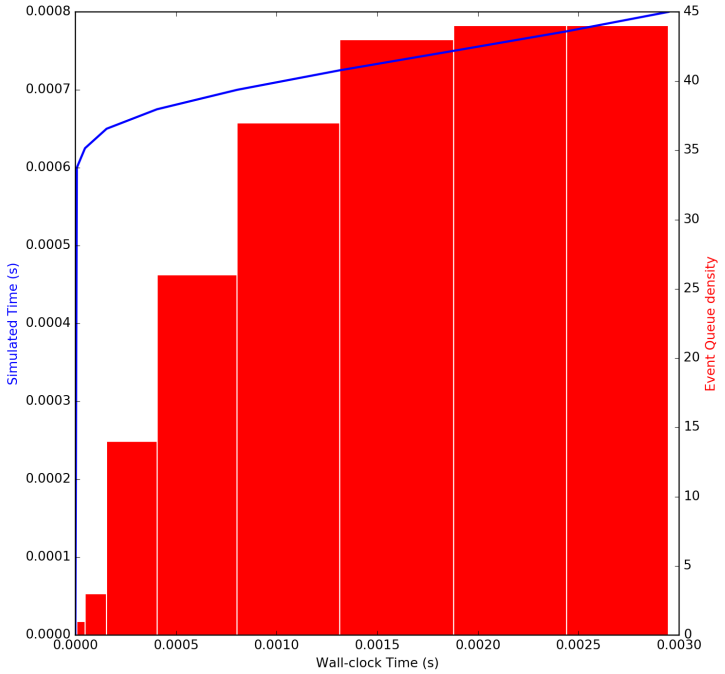


Figure 5.8: Simulated time and event queue size as a function of wall-clock time for the 48 node simulation run (Figure 5.7).

clear that messages are multiplied as greater numbers of nodes receive messages themselves. This expectation is proved by the increasing number of events in the queue as the simulation progresses.

Figure 5.9 shows the same information as Figure 5.8 but instead plots the event queue size against *simulated* time instead of the wall-clock. The initial delay of 600μs is parametrisable to simulate the communication lag between the target SpiNNaker system and the host computer. After this point, events can be seen multiplying in the same pattern. The widths of the bars here are uniform because all messages are assumed to require the same transmission and processing time. This is acceptable as only NN communication is used and the propagation delay between nodes is uniform.

Figure 5.10 shows how the simulation scales as the fault-map increases the number of discoverable nodes in the machine model. It is clearly linear in the number of events that are generated hence it follows that the wall-clock time also increases linearly. Usually, it is more important to show how the *simulated* time changes as the problem size increases, but this is only appropriate for algorithms whose completion can be detected.

This is not achievable with Algorithm 5.1 because there is no obvious method for reporting that all nodes have been assigned identifiers—long-range (i.e., P2P) communications have not yet been initialised. Furthermore, the total number of nodes in the system cannot be accurately known. Multiplying the extents W and H is only sensible if the lattice

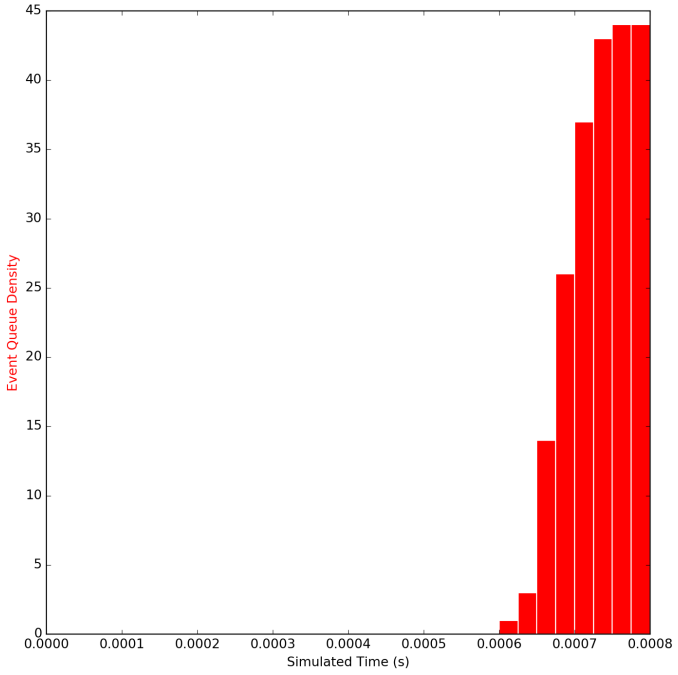


Figure 5.9: Event queue size for each simulated time-step for the 48 node simulation run (Figure 5.7).

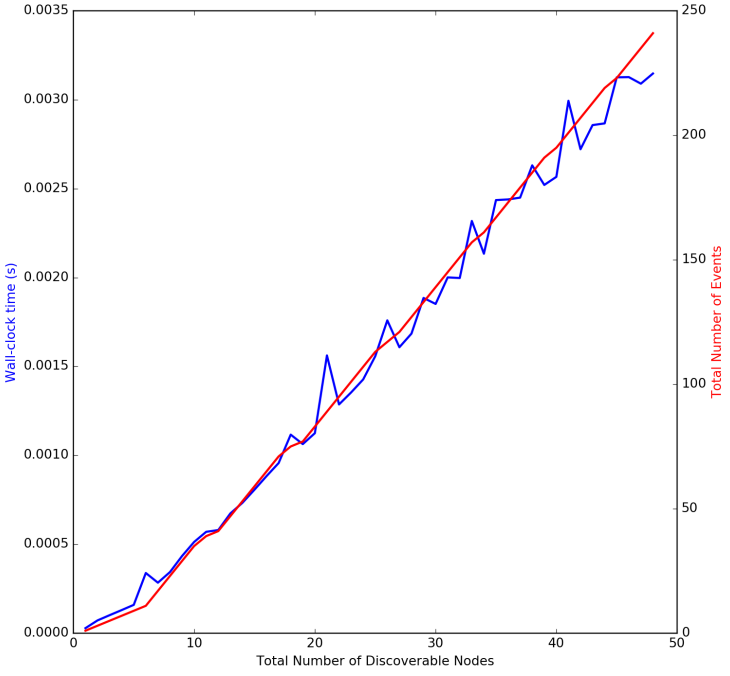


Figure 5.10: Simulator wall-clock time and total number of events for increasing numbers of enabled nodes.

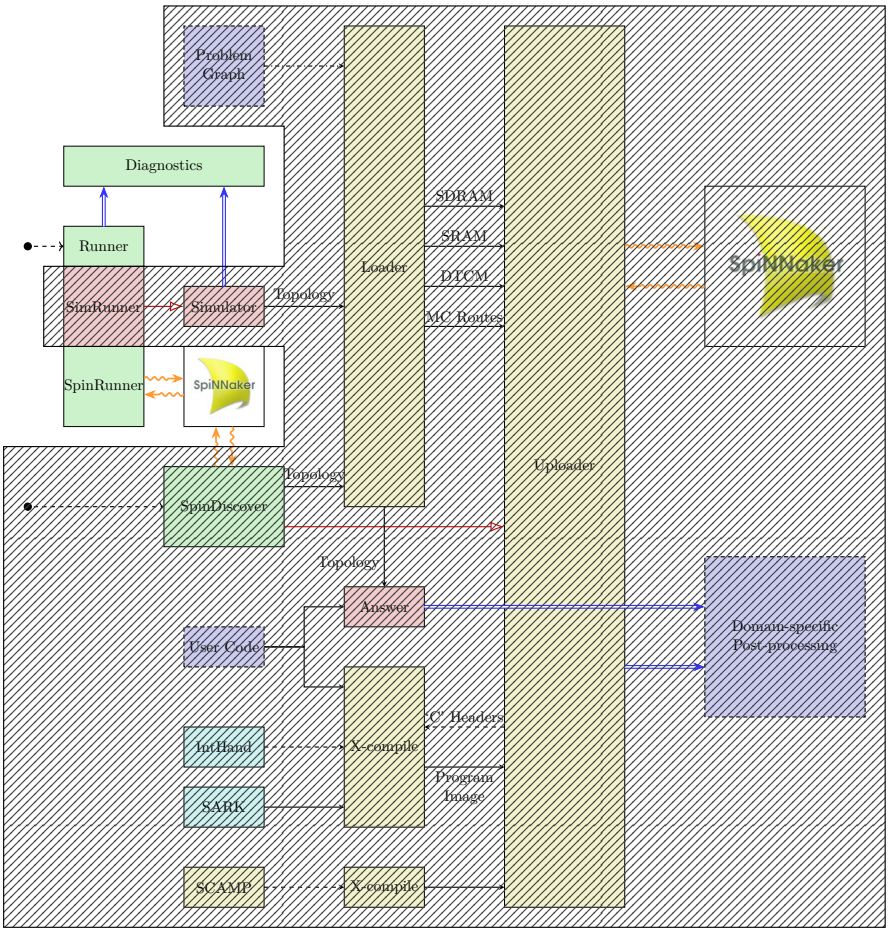


Figure 5.11: Stages of the toolflow shown in Figure 4.1 required to implement and verify the auto-discovery algorithms on SpiNNaker.

is complete, which is not the case for the SpiNN-103 topology. In physical SpiNNaker hardware, termination is achieved through means of a time-out calculated as the worst-case time required to complete the process. Comparing the total simulated time against the measured wall-clock time of SpiNNaker is therefore meaningless as it will be roughly equal to the time-out in every case.

5.4 Implementation

Figure 5.11 highlights the part of the tool-flow developed in this section, which targets SpiNNaker instead of the Simulator block. Much of the infrastructure is the same because Runner defers the running of experiments an `Executive` object which handles the specifics of the target platform. In the previous section, the `Executive` used, `SimExecutive`, targets the Simulator block instead of SpiNNaker. In this section, `SpinExecutive` is used to target a physical SpiNNaker machine running a modified version of SCAMP (see section 3.2). From the perspective of Runner, the experimental procedure (including fault-map generation) remains the same.

5.4.1 SCAMP Modifications

For the SpiNNaker experiments in this chapter, SCAMP is modified to understand an extended set of SCP commands which enable and disable other hooks that have also been introduced. `SpinExecutive` makes use of the Uploader API (the relationship shown in Figure 4.1) to transmit this range of SCP commands to SCAMP and receive responses containing any appropriate experimental results (such as discovered topologies and packet flow statistics).

The flood-fill process described in section 3.2.1 still applies to the modified image because any extensions that would affect the transmission of NN or P2P messages are initially disabled. The extended set of SCP commands are always available, which allows the `SpinExecutive` object to enable the modifications as required.

Several hooks are installed into the standard SCAMP executable:

SCP handler Extended commands to support `SpinExecutive` as mentioned above.

NN handler When enabled, all NN interrupts are processed by the extension code instead of the default SCAMP implementation. All SCAMP features that require NN messages cease to work in this state, which is why this hook is initially disabled. Once enabled by an SCP command over Ethernet, it is flood-filled to the rest of the system using the standard NN flood-fill technique built into SCAMP. It can be disabled again using a similar system built into the extension code, meaning that after the algorithms in this chapter have configured a SpiNNaker system, it can be restored to a state that can support applications.

100Hz tick handler SCAMP uses a handler called at a rate of approximately 100Hz to action system-level activities such as time-outs (e.g., for the standard P2P initialisation described in Chapter 3) and to spread out network traffic temporally in operations such as the standard P2P table construction implementation. With NN handling disabled, this handler can cause additional traffic to be injected into the network that could negatively impact any experiments being performed. Therefore, this handler is overridden when the NN handler is overridden, and restored when it is restored.

As with the hardware model used by the simulator, ports can have state and parameters assigned against them, and node behaviours are implemented as state-machines. Most of the discovery algorithms assign nodes with identifiers that are used to eventually build the P2P routing tables. This state is deliberately isolated from the state of SCAMP so that algorithms (implemented as behaviours) can be tested without affecting the general operation of SCAMP.

Isolated state can be copied into hardware registers and the SCAMP core afterwards if required. The algorithms designed in this chapter form the revised boot-sequence orchestrated by SpinDiscover (see Figure 4.1) which will be described in the following chapter. Once the topology has been discovered, SpinDiscover issues an SCP command that performs the state copy, allowing the node to function as expected but using the configuration constructed by the algorithms in this chapter.

5.4.2 Port Survey—The α-ping

When the simulator constructs a machine model from the input topology, port faults are determined by edges that are missing after a SpiNNaker stencil function is applied. In hardware, these faults are not (and cannot be) known *a priori* and must therefore be discovered before any algorithm experiment runs can be performed. This is a requirement of the hardware model established earlier where algorithms should not individually be expected to detect faulty ports.

Faulty ports are discovered by the α-ping which uses the tokens **request**, **response**, and **timeout**. All nodes begin in the *idle* state and all ports begin in the unknown state. The external host initiates the process by passing an initial **request** over the Ethernet connection, which causes this node to consider itself the *root*. As it is the root, it starts a timer that will lead to a **timeout** token being emitted from *all* ports that have been found to be functional. Each node implements the state machine shown as Algorithm 5.2. When the timer on the root node expires, the behaviour will be as if a **timeout** token has been issued to itself. However, in the implementation itself this is unnecessary and replaced instead by a function call.

In Algorithm 5.2, all state(p) assignments are assumed to act on the *physical* state of the port. All other algorithms may only adjust the *virtual* state of a port.

As with the P2P identifier assignment (Algorithm 5.1), completion cannot be detected and hence measuring the time taken to complete the process is not useful. The traffic flowing through the SpiNN-103 machine is shown in Figure 5.12. Each node contains a small pie-chart showing the ratio of local, non-local, and dropped packets as reported by the diagnostics registers in the router. The bar chart below the matrix shows the exact values for each node. Additionally, the results of the port survey are shown clearly on the visualisation into which the pie-charts are embedded.

Local and *non-local* refer to the sources of packets as they enter the router. Therefore local packets are those produced by a core within the node and non-local packets are those from outside the node. Dropped packets are caused by back-pressure being too great in the communications NoC. As expected, all non-edge nodes have equal amounts of local and non-local traffic because every **request** is met with a corresponding **response**.

Algorithm 5.2 α-ping port survey.

Require: All ports to be assigned the unknown state.
Require: Node state set to *idle*.

```
1:  procedure NNHANDLER(p_s, type)
2:      if node.state = terminal then
3:          return
4:      end if

5:      if type = request then
6:          if node.state = idle then
7:              node.state ← active

8:              for p ∈ node.ports; p ≠ p_s do
9:                  SENDNN(p, request)
10:             end for
11:         end if

12:         state(p_s) ← inbound
13:         SENDNN(p_s, response)
14:     else if type = response then
15:         state(p_s) ← bidirectional
16:     else if type = timeout then
17:         node.state ← terminal

18:         for p ∈ node.ports do
19:             if state(p) = unknown then
20:                 state(p) ← disabled
21:             else if state(p) = bidirectional then
22:                 SENDNN(p, timeout)
23:             end if
24:         end for
25:     end if
26: end procedure
```

Edge nodes have dropped packets or unequal numbers of local and non-local packets. The obvious expectation would be that all edge nodes would have dropped packets equal to the number of disconnected ports, which is clearly not the case. This is because the communications NoC includes a number of buffer stages to help reduce back-pressure. As this space is finite, the router is capable of issuing some packets to these ports without needing to drop them. However, it is not (nor can it be) obvious when packets are dropped because traffic prior to the SCP override command could still be resident in these queues.

The important results shown in Figure 5.12 are that all expected faults are correctly detected, and that the router diagnostics register data is successfully obtained from the machine.

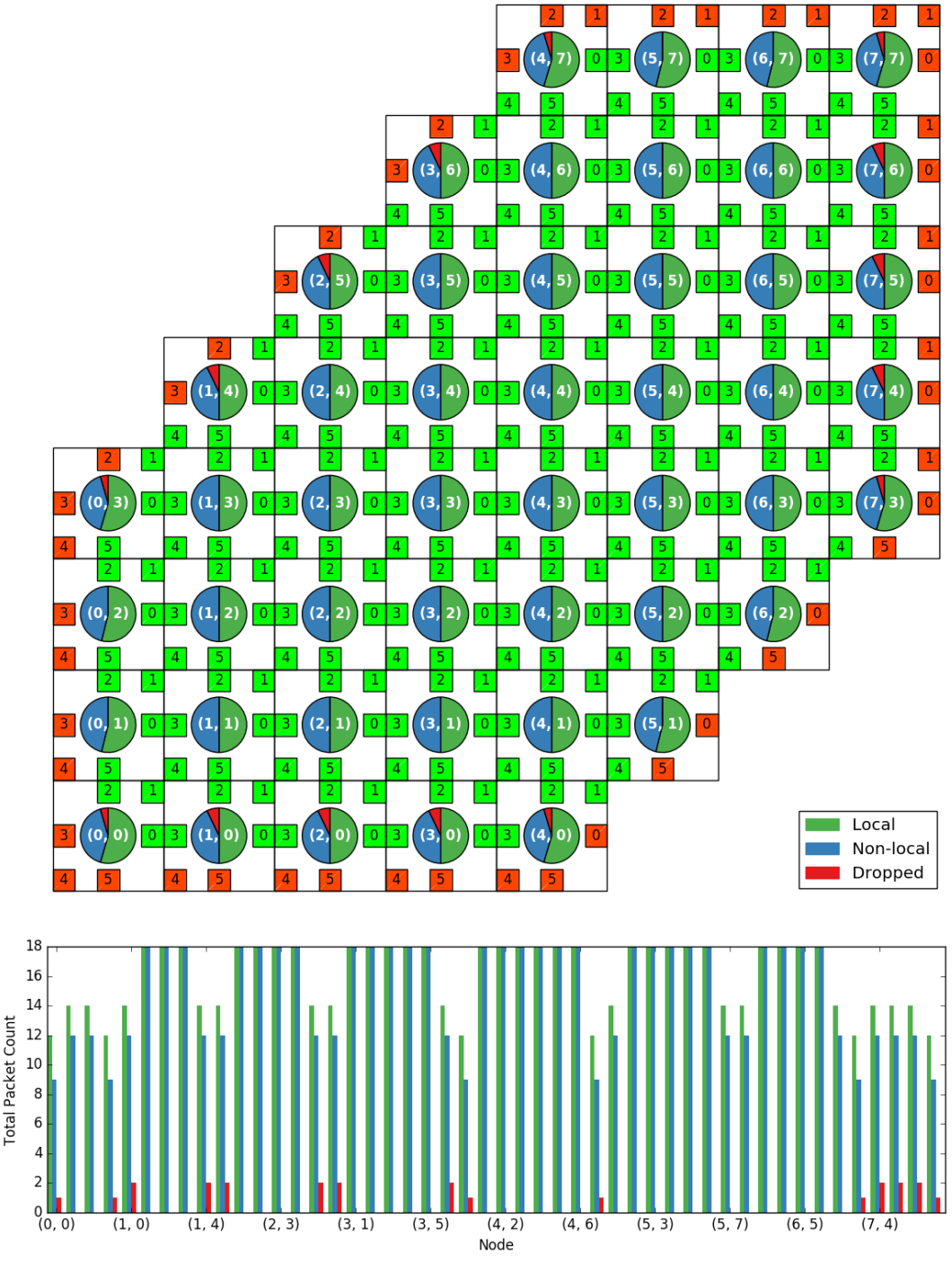


Figure 5.12: Distribution of packets across a SpiNN-103 machine for the α-ping.

5.4.3 Test Case

The results of the port survey essentially serve the same purpose as the machine model construction step required by the simulator. Now that the physical states of all ports

are known, the SpiNNaker implementation of Algorithm 5.1 can execute in the same manner as the behavioural implementation under the simulator.

Fault maps are generated in the same manner as before, but Runner applies them to the target SpiNN-103 hardware via the `SpinExecutive` which sends the required port states directly to each node using SCP commands. Clearly this requires that all nodes are addressable already. The default SCAMP P2P configuration step explained in section 3.2.1 is executed first to assign the node identifiers and construct the P2P tables.

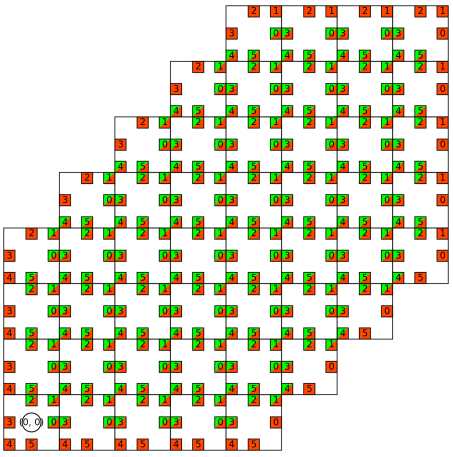
Whilst it may seem counter-productive to use the original implementation of the Algorithm 5.1 to configure the machine, consider that a) Figure 5.12 has shown there to be no port faults on the SpiNN-103 system being used, and b) the purpose of the implementation of Algorithm 5.1 is to validate the experimentation environment shown in Figures 5.1 and 5.11 so that it can be used to prove the algorithms designed in this chapter. These are then assembled into a single tool, SpinDiscover, that has no reliance on `p2pc` at all.

Figure 5.13 shows the same range of fault-maps shown in Figure 5.6 but instead applied to the SpiNN-103. The same stepping is used for ease of comparison, which clearly shows the same results between simulation and SpiNNaker implementation—the figures are identical.

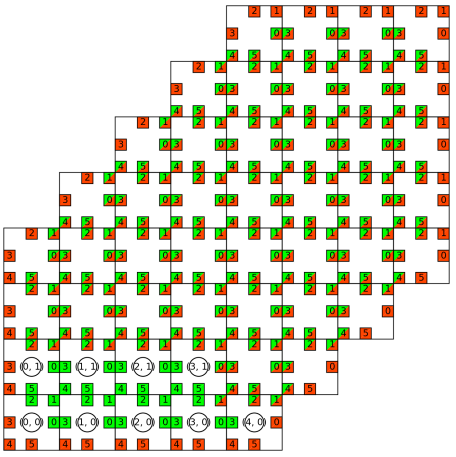
SpiNNaker diagnostics do not permit the same event-level plots that were shown for the simulation run. However, the router diagnostic registers allow for Figure 5.14 to show the traffic distribution over the course of the experiment run. The pattern is indicative of Algorithm 5.1 as packets are only ever emitted from the eastern, north-eastern, and northern ports.

Non-edge nodes have equal numbers of local and non-local packets because all six ports are functional; i.e., messages are received on the western, south-western, and southern ports, and new messages are transmitted through the others. Edge nodes vary slightly based on the number of ports that are capable of transmitting and receiving messages, but this proportion is correct in all nodes. Node $(0, 0)$ cannot receive any messages and has 100% local messages as expected. Similarly node $(7, 7)$ cannot transmit, and therefore has 100% non-local messages as expected. No messages have been dropped because only known-good ports have been used and the traffic density at all points has not been high enough to apply sufficient back-pressure to induce drops.

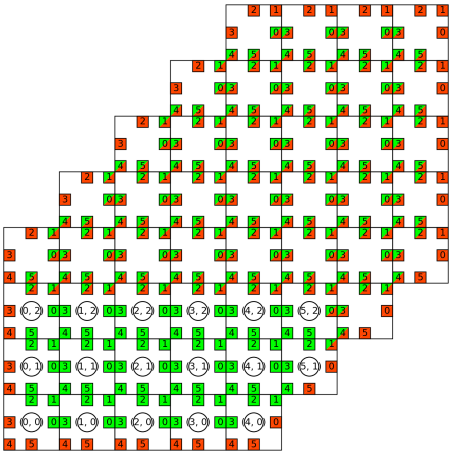
Inspection of the bar chart in Figure 5.14 indicates that double the number of packets implied by Algorithm 5.1 have been sent. This is due to a *terminal* state being introduced which allows the nodes to reset after the time-out has expired so that further experiments can be performed. After the time-out expires on the root node, a **timeout** token is sent



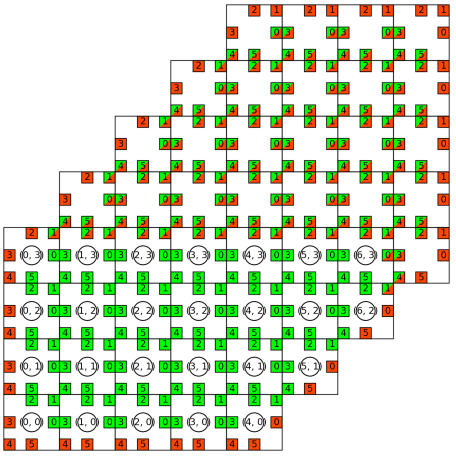
(a) Single node enabled.



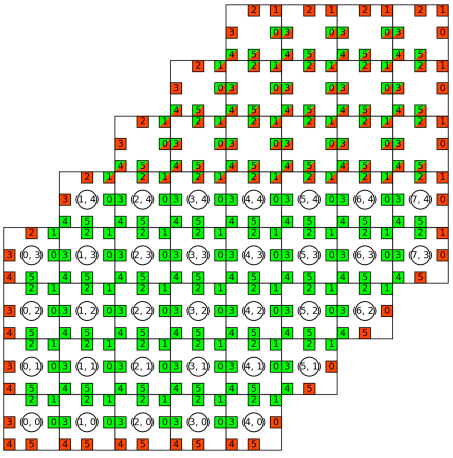
(b) 9 nodes enabled.



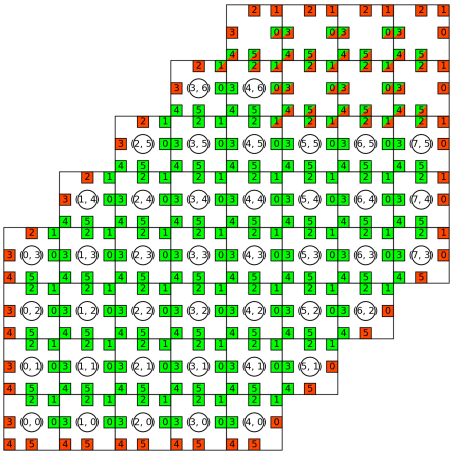
(c) 17 nodes enabled.



(d) 25 nodes enabled.



(e) 33 nodes enabled.



(f) 41 nodes enabled.

Figure 5.13: Machine state downloaded from a SpiNN-103 machine after being subject to various fault-maps as with the simulator in Figure 5.6.

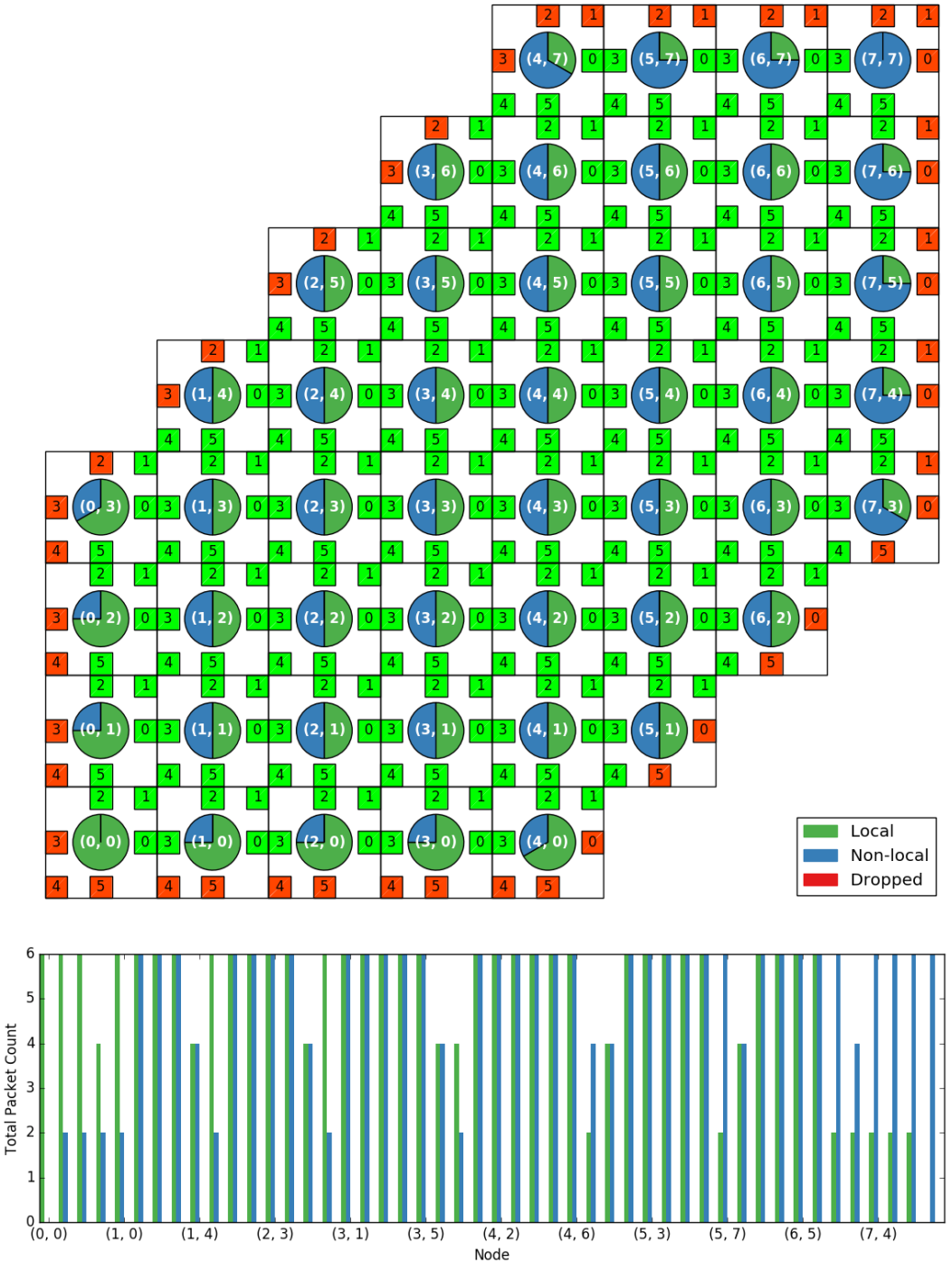


Figure 5.14: Distribution of packets across a SpiNN-103 machine for the P2P identifier assignment algorithm (Algorithm 5.1).

in the same pattern as the identifier tokens. This has the effect of doubling the number of messages without affecting the overall pattern of transmission.

Both test cases demonstrate the validity of the experimentation environment by showing that data can be collected and that fault-maps can be used to constrain the number

of discoverable nodes in both environments. The analysis code shared between both front-ends produces the correct plots, and both algorithms yield the same results. Time predictions from the simulator and time measurements from SpiNNaker are both equally meaningless for Algorithms 5.1 and 5.2 and hence have not been compared. However, this is not true of the algorithms designed in the following section and hence times can be compared.

5.5 Labelling Nodes and Building the Control Tree

Section 5.1 highlighted three key issues that must be addressed to support non-neural applications on SpiNNaker. The first of these issues, discovering hardware faults of the target system, is addressed by the α-ping introduced in the previous section. A suitable experimentation environment has been designed and validated in the previous few sections that allows this section to develop algorithms that address the two remaining issues.

Depth- and breadth-first searches are algorithms commonly used to visit all nodes of an arbitrary graph. The approaches developed in this section are all fundamentally based on either of these algorithms, but the visitation function establishes both a parent/child hierarchy to embed the control tree, and assigns contiguous (thus, unique) labels to each node.

Whilst it is not essential to perform a labelling step to discover the topology of the system, it does address an outstanding issue in Algorithm 5.1 that can lead to functional nodes being excluded under the fault conditions shown in Figure 5.15.

Figure 5.15 shows the machine model state after Algorithm 5.1 is run under the simulator. The topology is appropriately adjusted to remove the eastern (0) and northern (2) links from the root node. Despite only two port malfunctions existing, seven functional nodes have not been visited correctly. Clearly if this topology was toroidal, this issue could not arise. Indeed this issue has not arisen on any SpiNNaker hardware to date, but it would be naive to assume that it will *never* happen over the life-span of a machine of this scale.

Figure 5.16 shows the same fault conditions being applied to a SpiNNaker run of Algorithm 5.1, but instead making use of virtual port state instead of removing the topological links as this is not feasible on physical hardware. By inspection, it is clear that the problem is reproducible both under simulation and on hardware.

The topological discovery algorithms designed in this chapter necessarily visit each node in the system anyway so that a full picture of the connectivity can be created. Introducing label assignment into this is a trivial extension that completely removes the above

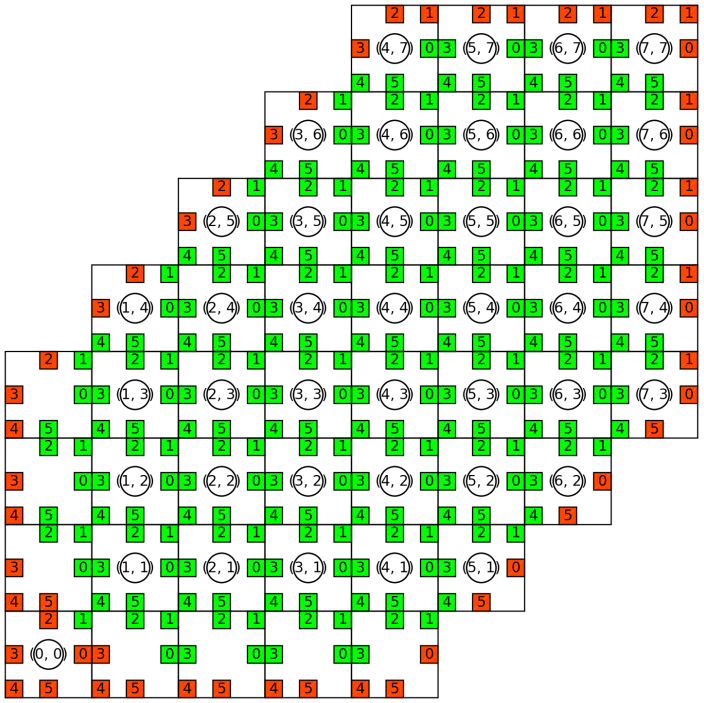


Figure 5.15: Simulation result demonstrating the potential issue with Algorithm 5.1.

fault condition *for all applications*, not just those in the non-neural domain. Therefore, the algorithms that follow should be considered a replacement for Algorithm 5.1 in addition to discovering the topology of a target SpiNNaker machine.

5.5.1 Depth-first Labelling

Back-pressure presents an unknown problem because of the non-deterministic nature of the distributed buffers of the communications NoC. This can be trivially ignored by limiting to the number of packets being transmitted at any instant to just one. Clearly this makes poor use of massively-parallel hardware, but it is guaranteed to lead to the correct result. The α-ping has already determined the physical state of the hardware, which removes any potential stalls due to faulty hardware.

A *depth-first* search naturally limits the packet traffic because the traversal attempts to find the longest possible path through the graph. Multiple searches cannot be performed in parallel as packets could collide. The algorithm requires the following tokens:

- **label**(k): Sent to an unknown topological sibling node in an effort to claim it as a child node, and assign it the label, k.

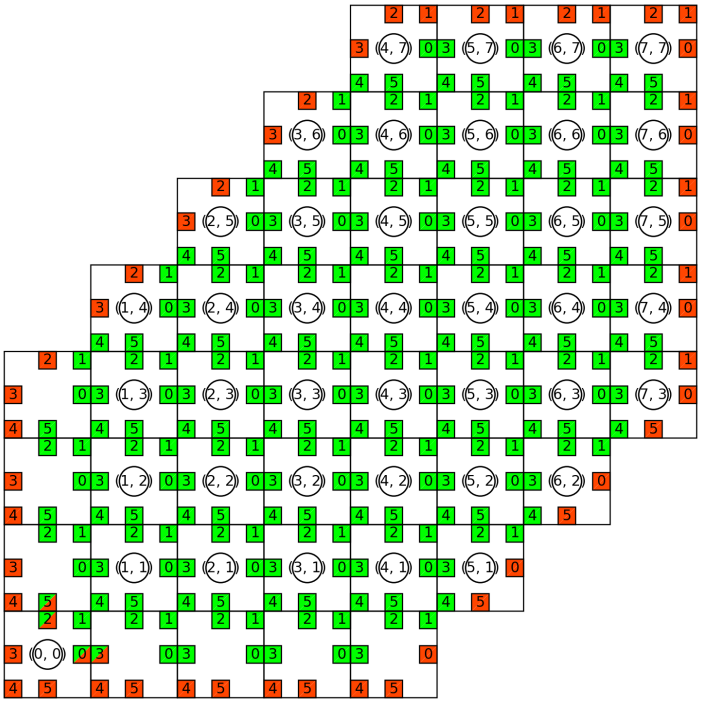


Figure 5.16: Result from SpiNNaker implementation demonstrating the same potential issue with Algorithm 5.1 shown in Figure 5.15.

- **ack**(k'): Response to a **label**(k) indicating that the child request has been accepted and that the topological siblings of this node have also been visited. k' is the highest identifier assigned to any child of the chain; for leaf nodes, $k = k'$.
- **nack**: Response to a **label**(k) indicating that the child request has been refused because the node already has a parent.
- **term**(N): Terminal message requiring that the nodes enter their barrier immediately to prepare for any subsequent behaviours. N is the total number of nodes discovered in the system.

Additionally, the following states are required:

- *idle*: node is waiting for messages from other nodes. Upon receiving a **label**(k), the node enters *labelling* and marks the inbound port as its parent.
- *labelling*: node has been claimed as a child by a sibling and is attempting to claim other siblings as its children. Upon entering this state, the node assigns itself the label k and initialises a counter, $c = 0$. All ports that are bidirectional are then set to unvisited.

  The first unvisited port starting from east (i.e., port 0) is visited and is used to send **label**$(k + c + 1)$ token to a potential child. If the sibling responds with **ack**(k'), c

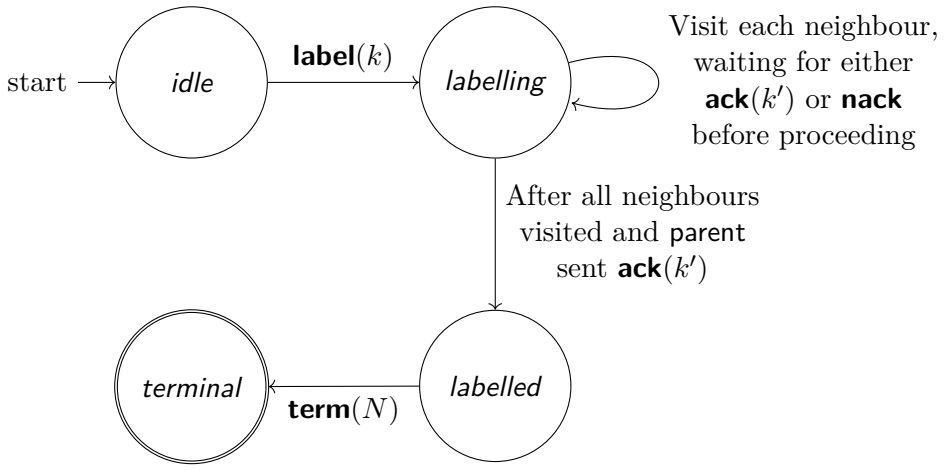


Figure 5.17: State transition diagram for the depth-first discovery algorithm.

is incremented to generate a new label and the port is marked as child. After a response of either **ack**(k') or **nack**, the process is repeated with the next unvisited port.

After all ports have been visited, the node advances to the *labelled* state.

- *labelled*: node has finished labelling itself and all possible children. Upon entering this state, the node sends **ack**$(k + c)$ using the parent port.
- *terminal*: node is in the terminal state and may be used for other purposes. A node enters this state then receiving a **term**(N) token from its parent and then propagates the tokens to all of its children. N is stored locally as the total number of nodes in the system. Child nodes respond to their parents with a corresponding **term**(0) token to acknowledge the terminal state. Leaf nodes respond immediately because they have no children.

Nodes may only advance state as a response to a packet arrival (shown by Figure 5.17) with the single exception of the *root* node, which automatically advances into the *terminal* state once all of its siblings have either responded with **ack**(k') or **nack**. Once the terminal acknowledgements propagate back up to the root node, the process is complete. In simulation, this simply terminates the process, whereas on SpiNNaker an SCP message is emitted reporting the discovered size of the machine.

5.5.1.1 Simulation

Before implementing this behaviour on SpiNNaker, it is verified under simulation. As before, the algorithm is subject to the same fault-map trajectory to gain an estimation of how the algorithm scales with the number of discoverable nodes. Figure 5.18 shows that the simulated running time scales linearly with machine size, which is to be expected because only a single node is active at a time and all nodes *must* be visited.

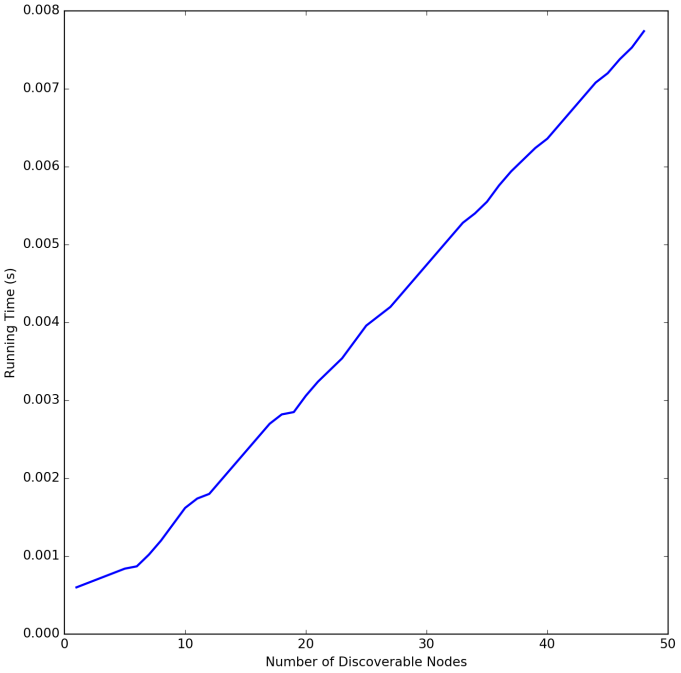


Figure 5.18: Scaling of the depth-first discovery algorithm with machine size under simulation.

Figure 5.19 provides an indication of the simulator performance for the 48 node simulation run. In this circumstance, the simulated time is a roughly linear function of wall-clock time, indicating that the simulator is running at approximately the same rate as the SpiNN-103 hardware. This is to be expected as only a single node of the SpiNN-103 is allowed to proceed at any instant, essentially reducing it to the performance of a single-threaded process.

Comparing Figures 5.19 and 5.20 shows that only a single message is in flight at a time, with the exception at around 7ms simulated time where it increases to two for a short period. This is caused by the terminal state barrier explained earlier. This is clearest in Figure 5.20 which colour-codes the events in the queue according to the type of message being carried. **label**(k), **ack**(k'), and **nack** tokens are all interleaved until about 6.5ms simulated time where the barrier transmission begins. Observe also that the first block of tokens are all **label**(k), which implies that a string of nodes are being claimed without any contention until around 1ms simulated time.

Both of these observations are confirmed by Figure 5.21. Firstly, nodes 0–18 form a long string without causing contention. Node 18 then attempts to claim 10, 11, and 12 before finally being able to claim what becomes node 19, leading to the first group of **nack** tokens in Figure 5.20. Secondly, there are exactly two leaf nodes, meaning there

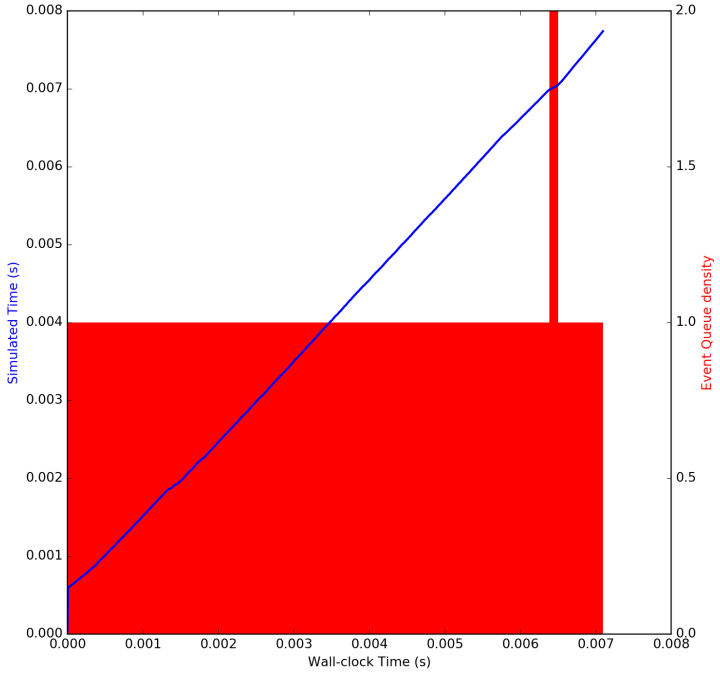


Figure 5.19: Simulated time advancement and event queue density as a function of wall-clock time for the 48 node depth-first simulation run.

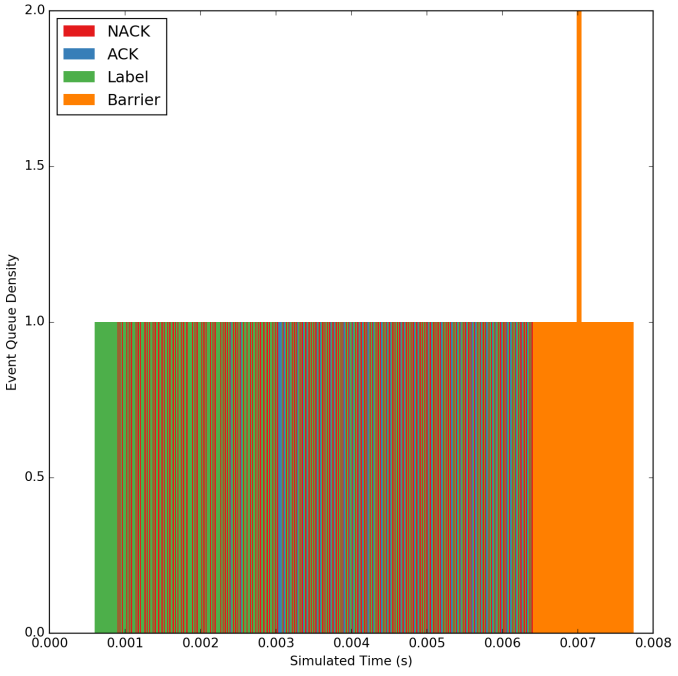


Figure 5.20: Distribution of event types (i.e., tokens) throughout the 48 node depth-first simulation run.

is a single branch which can be seen at node 39. This causes the the message count to increase to two because the barrier messages follow the embedded tree structure.

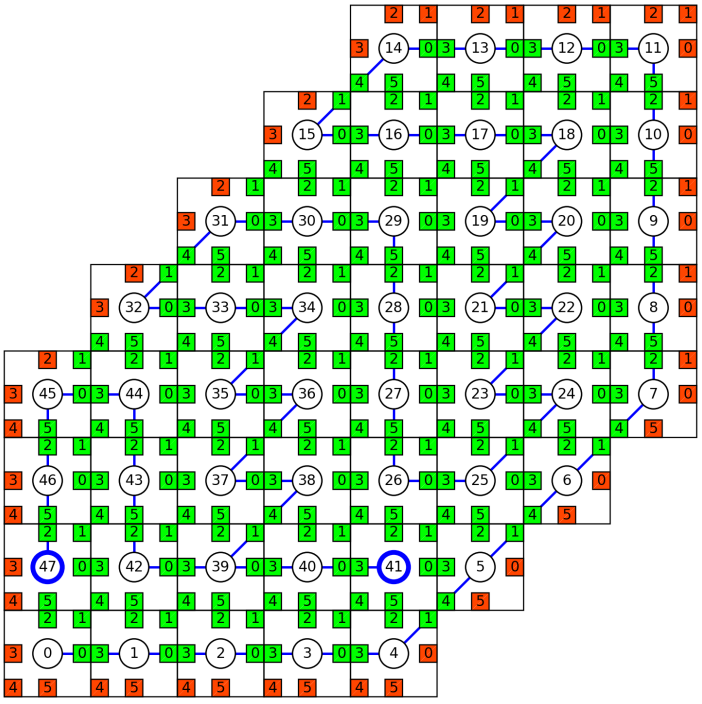


Figure 5.21: Discovered machine model state after the 48 node simulation run, showing the embedded tree and the assigned node labels.

Figure 5.21 illustrates that the labelling and tree construction stages have completed successfully (the leaf nodes are highlighted). Node labels form a contiguous set and, more importantly, are unique which will permit P2P table construction later on. Figure 5.22 subjects the depth-first approach to the same fault conditions described in section 5.3.3, and demonstrates that they can be overcome by this technique. This is expected because the search is designed for an *arbitrary* graph, so provided that at least one port is functioning (as is the case for node 0 in Figure 5.22), the complete graph will be discovered.

Both Figures 5.21 and 5.22 feature branches because transmission through ports is attempted in increasing index order. This also explains the zig-zag pattern exhibited in both figures. In Figure 5.21, the first branch appears at node 39 because it is the first node which comes into contact with an obstruction according to this rule. This happens much sooner in Figure 5.22 because the fault conditions imposed upon node 0 cause exploration to being on the second row of the grid instead of the first. Therefore, two branches develop at node 5 because of the grid boundary; the branch from port 1 forms a similar pattern to that of Figure 5.21, whilst the branch from port 4 is simply discovering the nodes that were originally obstructed by the fault.

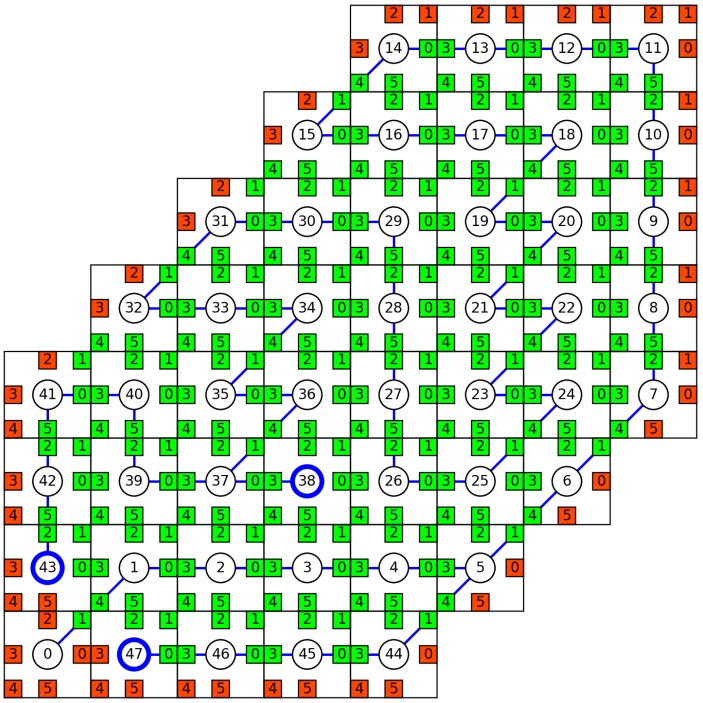


Figure 5.22: Resilience of the depth-first discovery simulation run against the same fault pattern as in Figures 5.15 and 5.16.

5.5.1.2 Implementation

Simulation results confirm that the depth-first labelling and discovery approach behaves as expected. Figure 5.23 shows the performance of the same approach implemented on SpiNNaker. The blue line shows the wall-clock measurements taken of the SpiNNaker implementation, whereas the red line shows the simulated time as predicted by the simulator after tuning the processing time parameter. They scale and perform almost identically, further validating the tool-chain and the approach itself.

The blue plot for SpiNNaker represents the average results from ten experimentation runs: the solid blue line is the median time measured for the various machine sizes, and the shaded light-blue region surrounding it represents the span between the upper and lower quartiles. Not only does the SpiNNaker implementation perform well and scale linearly, but the small shaded area shows that it performs consistently well. Extrapolating from Figure 5.23, the expected wall-clock time required to discover the SpiNN-105.5 system shown in Figure 3.18 is approximately 4.2s.

Figure 5.24 shows the traffic distribution across the SpiNN-103 hardware for the 48 node run. As expected, the packet flux is equal parts local and non-local traffic because only a single message is ever being sent at a time, and each request is met with a corresponding response. Coupled with faulty ports being naturally skipped by the behaviour described

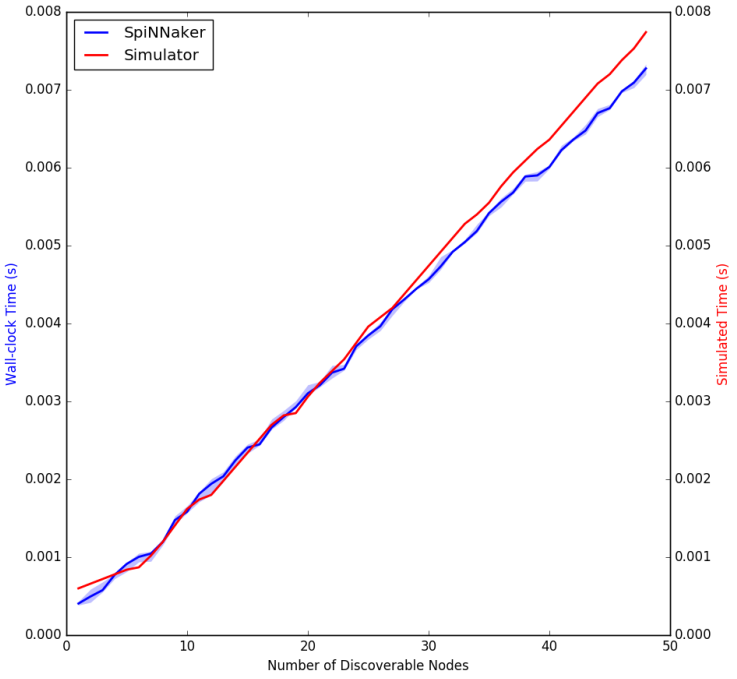


Figure 5.23: Comparison of the running time predicted by the simulator and the measured wall-clock time of 10 consecutive runs on SpiNN-103 hardware.

earlier, this eliminates any opportunities for back-pressure to occur, so no packets are ever dropped by the router.

The embedded tree structure is difficult to see in Figure 5.24 and is more clearly shown by Figure 5.25. This is identical to the simulation result shown in Figure 5.21 which is expected because the traffic pattern is entirely deterministic. As before, the leaf nodes have been highlighted to ease visual inspection of the structure. Additionally, the tolerance of the connectivity fault can be seen clearly in Figure 5.26, also resulting in the same structure observed earlier in simulation (Figure 5.22).

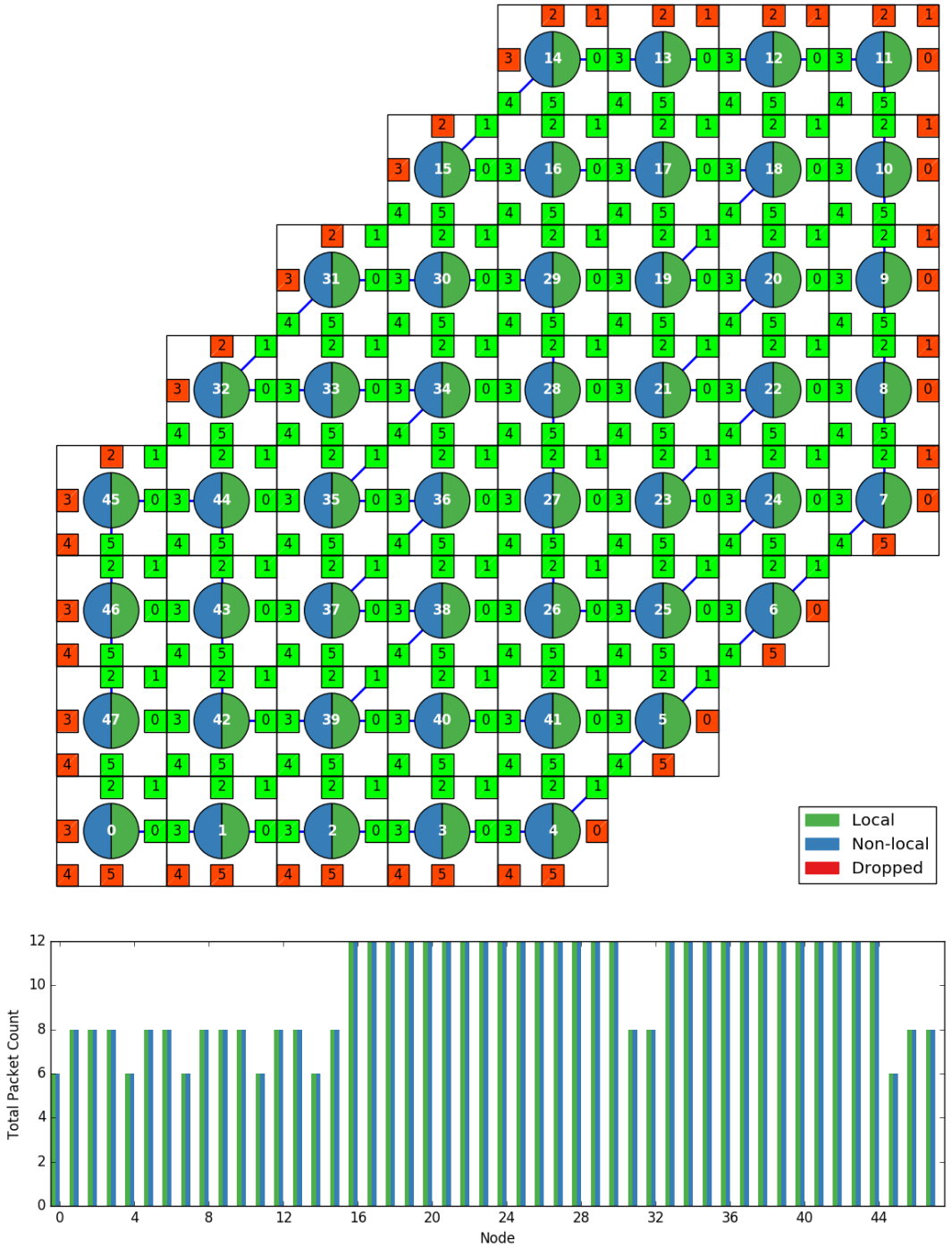


Figure 5.24: Packet distribution across the SpiNN-103 hardware for the 48 node run of the depth-first discovery run.

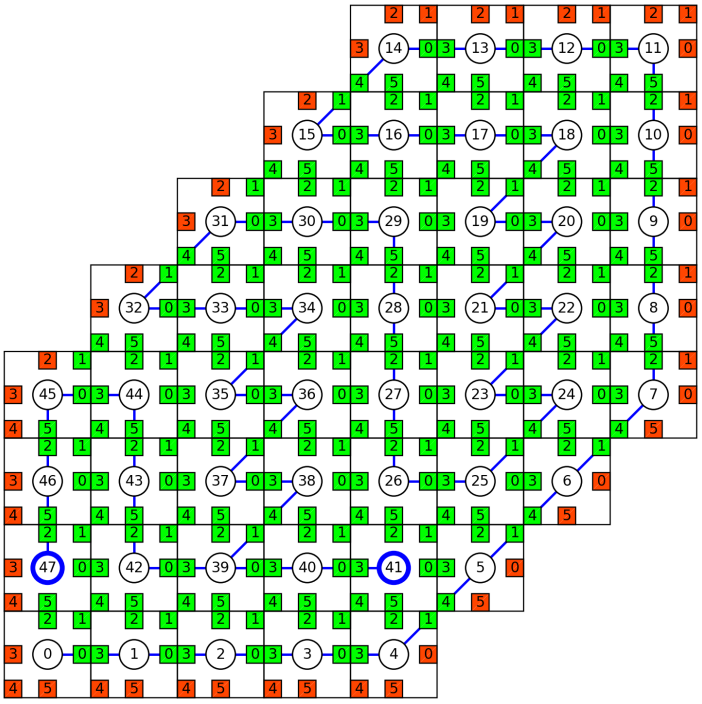


Figure 5.25: Discovered machine topology from the 48 node run on SpiNN-103 hardware.

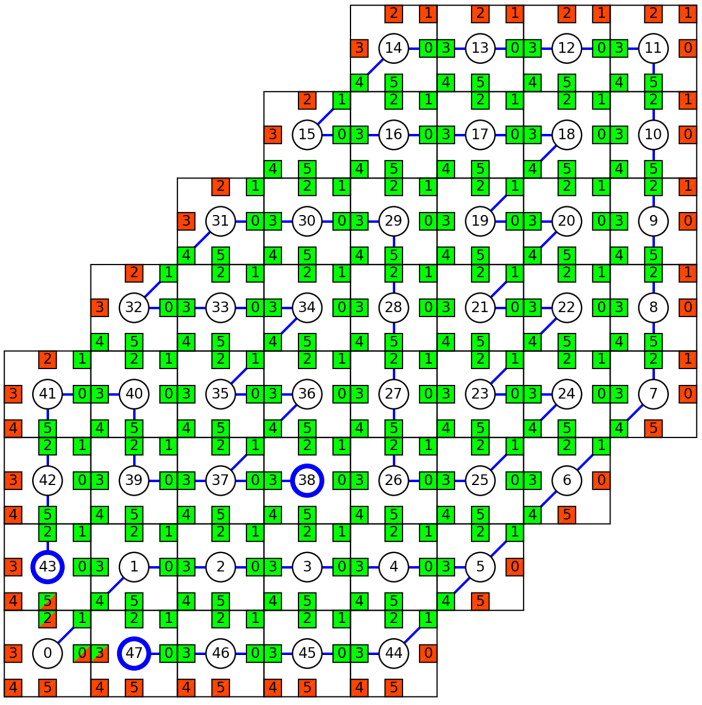


Figure 5.26: Resilience against the fault pattern of a 48 node run on SpiNN-103 hardware.

5.5.2 Lock-step Breadth-first Labelling

Whilst the depth-first search approach tolerates faults and visits every node in an arbitrary topology, it produces heavily skewed trees (Figures 5.21 and 5.25). Barrier releases following these trees will not be optimal (balanced trees require the shortest time) and could introduce significant delay in applications that make frequent use of this synchronisation technique. With a few additional tokens and states, the depth-first discovery algorithm presented in the previous section can be converted into a *breadth-first* algorithm whilst still requiring on a single packet in transit at a time.

The labelling state must be broken into two discrete operations: firstly, a node must choose to either accept or reject a label request from a neighbour node; secondly it must attempt to claim siblings as its own children. To support this, the following additional tokens are introduced:

- **label-children**(k): Sent to a node to start it claiming siblings as child nodes, or to allow previously claimed child nodes to do the same.
- **ack-children**(n): Response to a **label-children**(k) token reporting that n additional child nodes have been labelled by this request. This is the *total* number of new child nodes, not merely the siblings that have been converted into children by the request.

These tokens are used in addition to those introduced in the previous section for the depth-first algorithm. The following state changes are also made to alter the visitation pattern:

- *child*: a node receiving a **label**(k) token will enter this state from *idle* instead of entering *labelling* directly. The identifier in the token has accept, and a **ack** token is issued as the response. No siblings are visited at this point.
- *labelling*: is entered after a node in the *child* state receives a **label-children**(k) token. All unvisited ports are visited and issued with a **label**(k) token. k is calculated in the same manner as with depth first, and ports are visited in the same sequential manner.

  Once all ports have been visited and siblings claimed as children where appropriate, the state advances to *labelled* and a **ack** is issued to the parent reporting the number of siblings converted into children.
- *descendants*: a node enters this state from *labelled* upon receiving a **label-children**(k) token. A counter is reset, $c = 0$, and then each child node is issued with **label-children**$(k + c)$ in sequence starting with the child port with the lowest index. c is

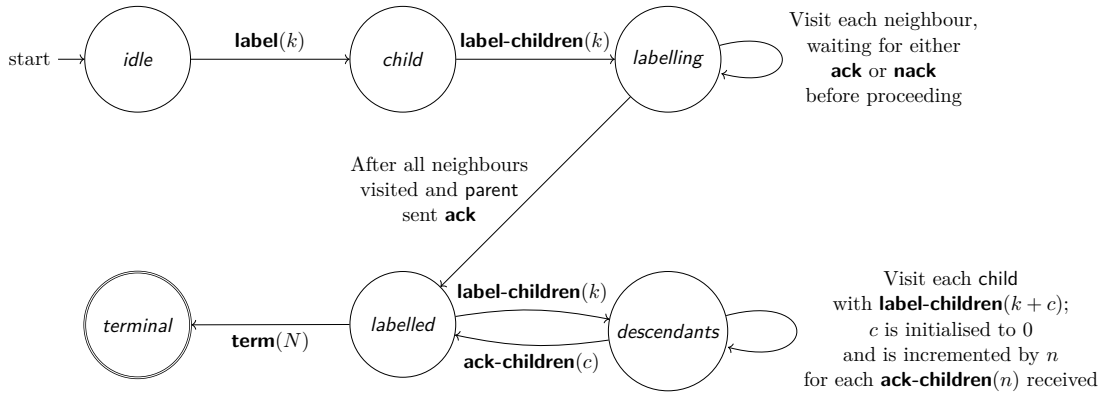


Figure 5.27: State transition diagram for the lock-step breadth-first discovery algorithm.

incremented by the n returned by each **ack-children**(n) token, causing new nodes to be allocated contiguous identifiers. Once all child nodes have been visited in this manner, the node responds to its own parent with **ack-children**(c), where c represents the total number of child nodes that have been claimed, which may be many levels lower in the hierarchy than the immediate child nodes.

Figure 5.27 shows the extended state transition diagram. As before, the root node advances through the states automatically to progress the algorithm. Once it has labelled its immediate children, it repeatedly oscillates between the *labelled* and *descendants* states until the sum of all **ack-children**(n) responses is zero, implying that no new child nodes have been added. Once this happens, the root will issue a **term**(N) to all children as before, flooding the system with total number of discovered nodes and stopping the algorithm to allow for new behaviours.

5.5.2.1 Simulation

Figure 5.28 shows how the simulation of the lock-step breadth first algorithm scales with the number of nodes. The estimated running time grows slightly sublinearly, which is expected because some nodes are visited multiple times. **term**(N) tokens can propagate with increased parallelism as the embedded tree is more balanced, which most likely accounts for the sublinear scaling.

Figure 5.29 shows how much more balanced the embedded control tree is because the significant gradient change in the simulated time against wall-clock time. Up until this point, the algorithm is running in a highly sequential manner, hence the roughly linear relationship between simulated and wall-clock time. When the event queue density increases because multiple nodes are emitting tokens in the same time-step, the amount of wall-clock time required to compute the updates necessarily increases.

Figure 5.30 shows a large spike after about 5ms simulated time, indicating the increased parallelism of the termination barrier. Prior to this point, the event queue only ever has a single event in it at a time, which is expected. The pattern of event types shows an increasing number of **ack-children**(n) tokens being passed around whilst the other types decrease. This is indicative of the oscillatory state changes of the root node described earlier.

Figure 5.31 confirms the increased balance of the tree compared to Figure 5.21. Nodes are not uniformly distributed between all branches because tokens are emitted from ports using a monotonically increasing index. This artificially assigns priority to lower numbered ports, causing an increase branch density on these ports. However, the balance is a significant improvement overall whilst still maintaining a low in-transit message count. The balance would be enhanced by visiting ports in a random order.

As expected, the breadth-first approach is tolerant of the fault condition demonstrated in Figure 5.32. The artificial priority bestowed by the port indices has the effect of increasing the number of leaf nodes as the **label**(k) tokens expand from the root node in a radial pattern.

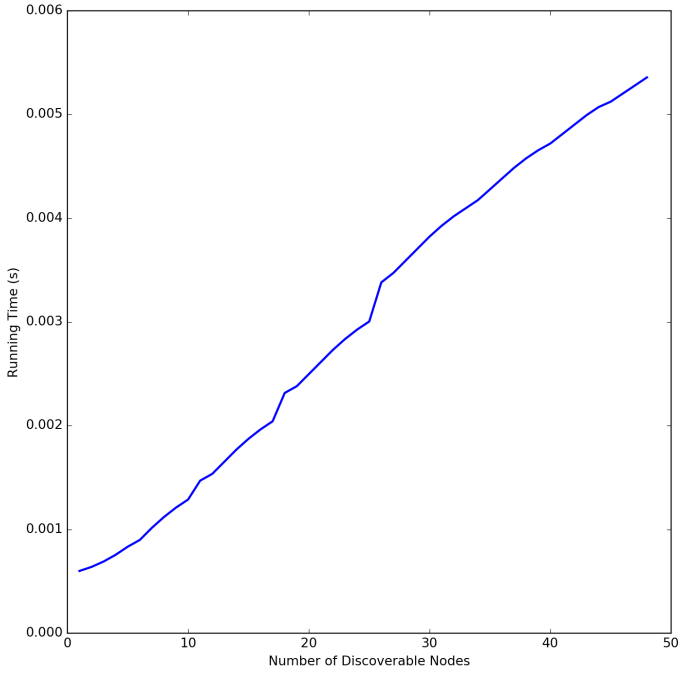


Figure 5.28: Scaling of the breadth-first discovery algorithm with machine size under simulation.

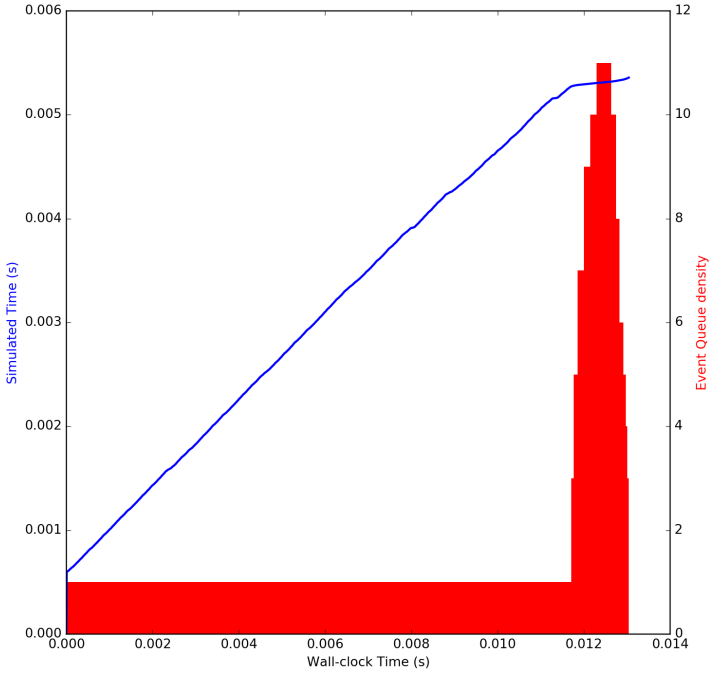


Figure 5.29: Simulated time advancement and event queue density as a function of wall-clock time for the 48 node breadth-first simulation run.

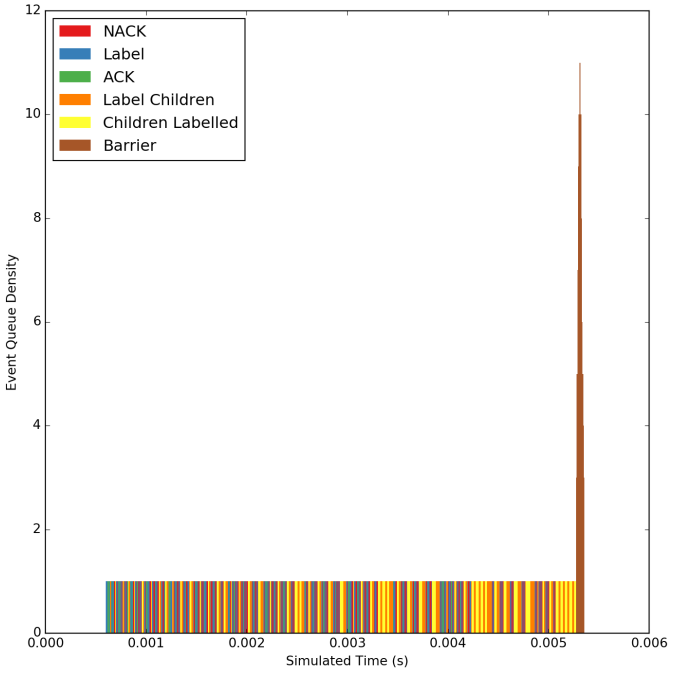


Figure 5.30: Distribution of event types (i.e., tokens) throughout the 48 node breadth-first simulation run.

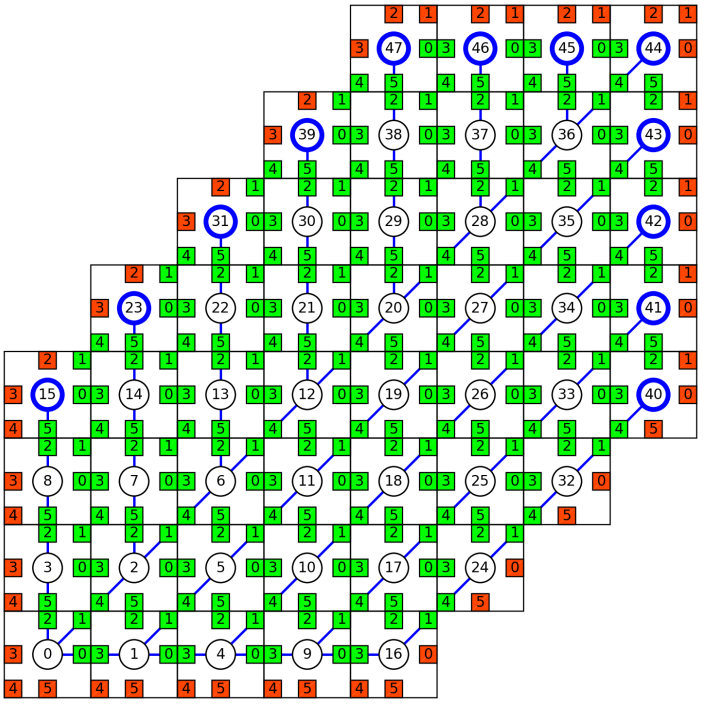


Figure 5.31: Discovered machine model state after the 48 node breadth-first simulation run, showing the embedded tree and the assigned node labels.

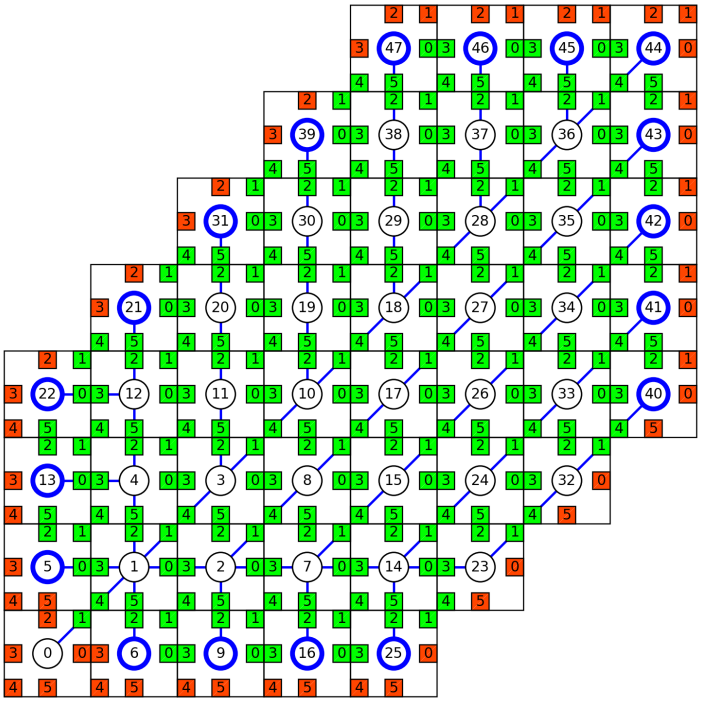


Figure 5.32: Resilience of the breadth-first discovery simulation run against the same fault pattern as in Figures 5.15 and 5.16.

5.5.2.2 Implementation

Figure 5.33 compares the predicted running time from the simulator after parameter tuning against the measured wall-clock time from the SpiNNaker implementation. The sublinear scaling is more pronounced on the SpiNNaker curve, but the variance is suitably low as with the depth-first result (Figure 5.23).

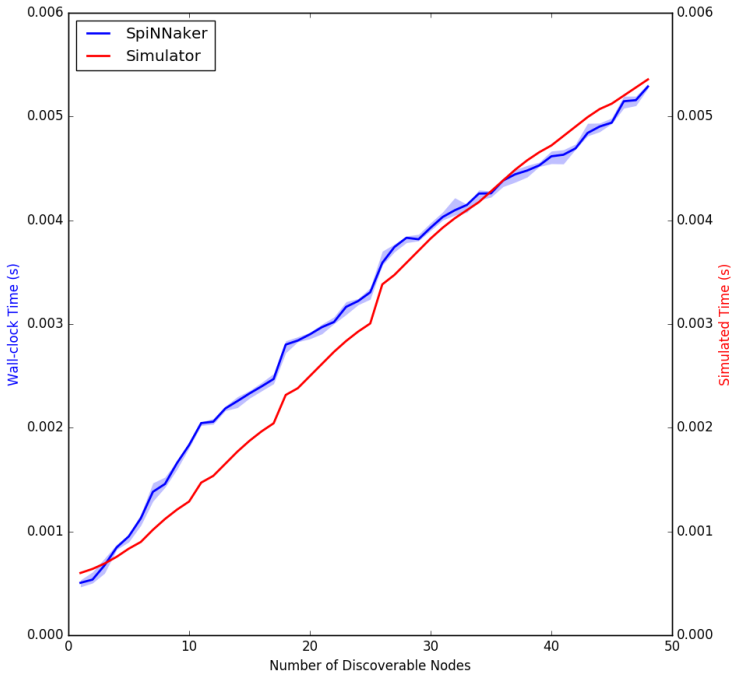


Figure 5.33: Comparison of the running times predicted by the simulator and the measured wall-clock time of 10 consecutive runs on SpiNN-103 hardware for the breadth-first discovery algorithm.

The traffic distribution pattern shown in Figure 5.34 steadily decreases for nodes farther away from the root. The oscillatory state changes described earlier cause a wavefront of tokens to expand from and subsequently collapse back to the root node each time new nodes are claimed as children. It follows that nodes with low indices will have to propagate greater numbers of tokens. As before, the ratio of local to non-local traffic is equal because every request is met with a corresponding response, and with the exception of the **term**(N) token propagation at the end of the algorithm, only a single message is being transmitted at a time.

Figures 5.35 and 5.36 show identical results to Figures 5.31 and 5.32 obtained from simulation, demonstrating that a (more) balanced tree can be reliably constructed and the fault conditions can be tolerated. As the sibling nodes are claimed one at a time, the pattern is entirely deterministic, so the simulation and SpiNNaker results are identical.

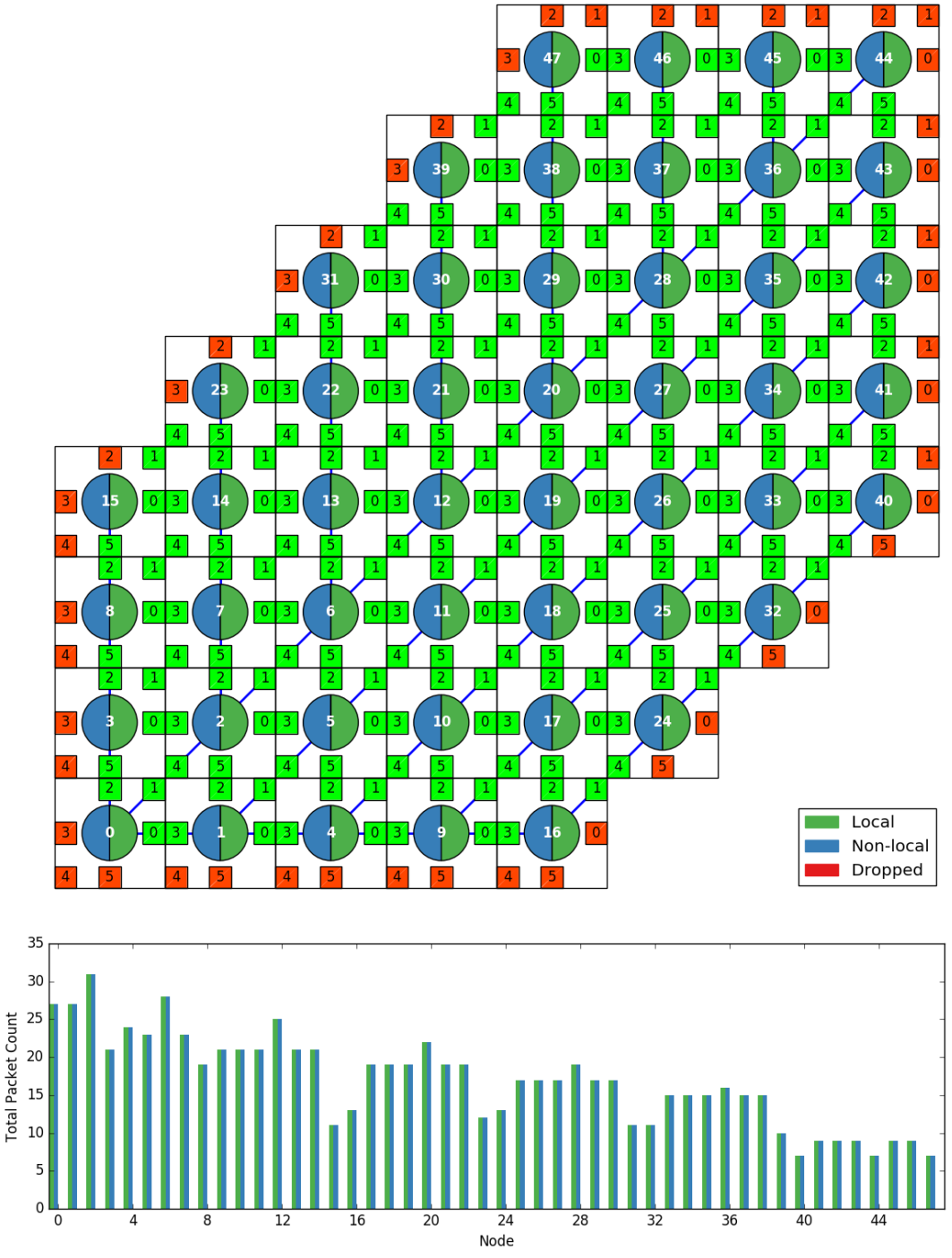


Figure 5.34: Packet distribution across the SpiNN-103 hardware for the 48 node run of the breadth-first discovery run.

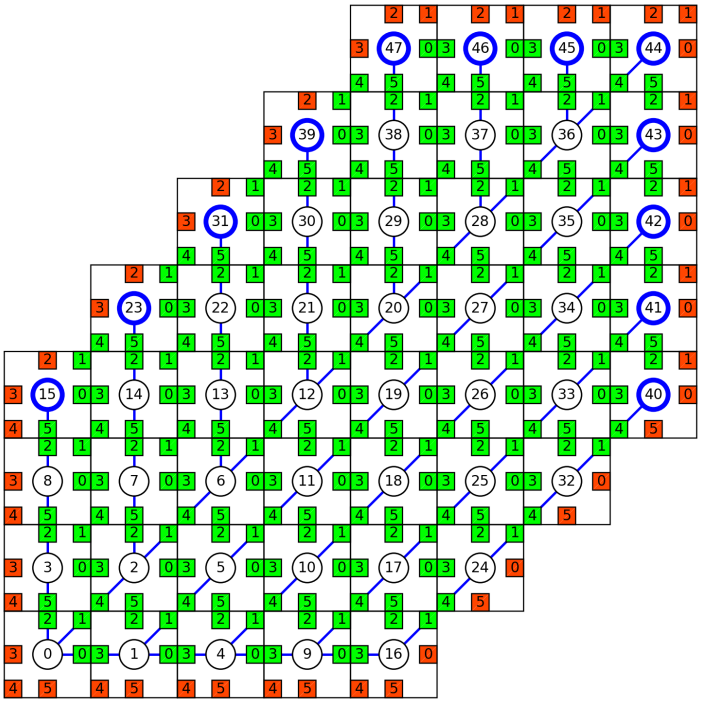


Figure 5.35: Discovered machine topology from the 48 node breadth-first run on SpiNN-103 hardware.

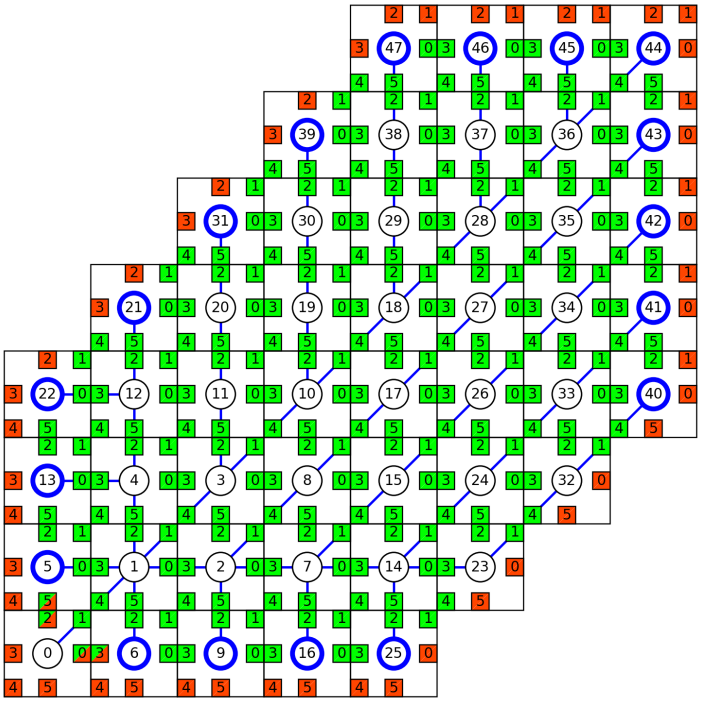


Figure 5.36: Resilience against the fault pattern of a 48 node breadth-first run on SpiNN-103 hardware.

5.5.3 Parallel Breadth-first Labelling

The two approaches introduced previously build the control tree and label all the nodes simultaneously, but they achieve this by limiting the volume of network traffic to just a single message in transit at a time. SpiNNaker is a massively parallel computing engine designed to support highly connected problems—making best use of the hardware means doing as much work in parallel as possible.

A parallel breadth-first discovery algorithm visits all the siblings of a node *at the same time* and relies on the network fabric to act as an arbiter. However, there is no reliable method of allocating node labels at the same time. Instead, the algorithm is divided into two stages: first, the control tree is established, then the control tree is walked to assign the labels. Whilst these stages must be performed in sequence, each stage is inherently parallel.

Four tokens are required:

- **claim**: Sent from a node to a sibling to claim it as a child. No payload is required because the purpose is purely to establish the hierarchy.
- **nack**: Response to a **claim** token to refuse the request.
- **ack**(n): Serves as the response to **claim** to accept the child claiming request, and to **label**(N, k_l, k_u) to mark that the labelling process (i.e., second phase) has completed.
- **label**(N, k_l, k_u): Begins the labelling phase on a portion of the established tree. As the tree is constructed, each branch is assigned to a specific port, and the number of nodes in that branch is stored against that port. k_l and k_u define the range of identifiers required by that particular branch. N is the total number of nodes discovered in the system.

The state machine uses the following states:

- *idle*: Default state for all nodes; upon receiving a **claim** token, the inbound port is marked as the parent and the state machine advances to the *claimed* state.
- *claimed*: Upon entering this state, a subsequent **claim** token is issued to all unvisited ports which are immediately marked as visited. All ports that respond with an **ack**(n) token are marked as child and n is stored as the *total* number of children accessible via that port. Once all requests have been matched by appropriate responses, an **ack**$(1 + n')$ token is sent via the parent port, where n' is the sum of all child nodes claimed. For leaf nodes, $n' = 0$.

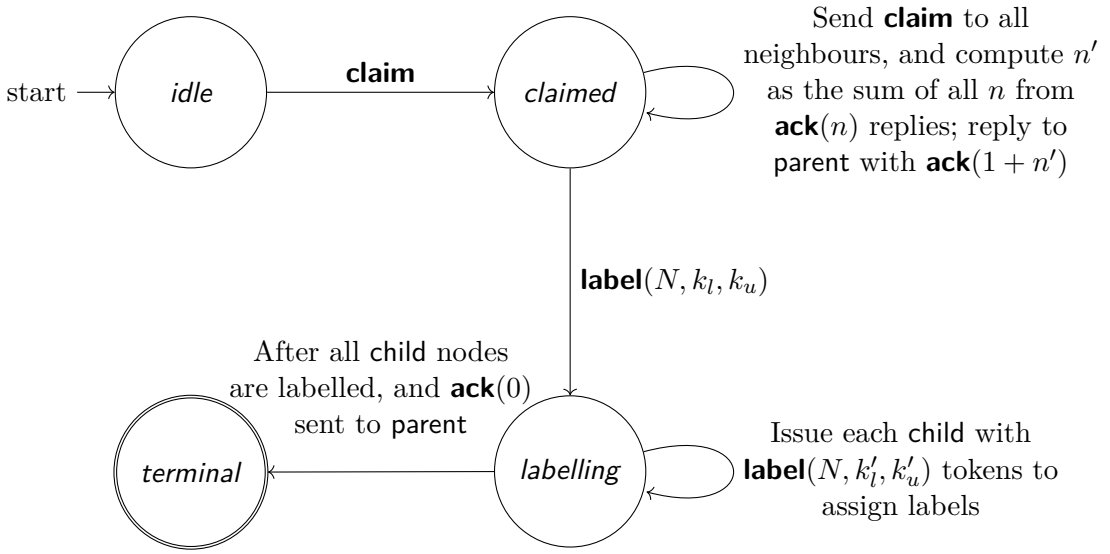


Figure 5.37: State transition diagram for the parallel breadth-first discovery algorithm.

- *labelling*: Nodes other than the root enter this state when receiving a **label**(N, k_l, k_u) token; the root node automatically enters this state after all of its siblings have returned an **ack**(n). $[k_l, k_u]$ defines the range of identifiers to use to label this branch of the tree. The current node accepts k_l as its identifier, then the rest of the interval is divided into appropriate sub-intervals issued to each child port. Once all child nodes have been labelled, the node automatically advances into the *terminal*. For leaf nodes, $k_l = k_u$.
- *terminal*: Upon entering this state, **ack**(0) is sent via the parent port to signal that all child nodes have been appropriately labelled. Leaf nodes do not have children and hence report to their parent immediately. Once the root node enters this state, it reports the total machine size (the sum of all its child port branch sizes) to the external host machine.

Figure 5.37 shows the state transition diagram for this algorithm. The root node automatically advances into *claimed* to begin the process, after which tokens propagate across the machine in parallel.

5.5.3.1 Simulation

As this approach embraces the parallelism of the platform, it scales non-linearly as shown in Figure 5.38. The multiple plateaus shown in this plot are particularly indicative of the fault-map generation pattern and directly correspond to the number of "rows" that have been enabled. The width of each "row" is irrelevant, meaning that maximum efficiency

is achieved with wide rows, making this approach particularly useful for large SpiNNaker machines.

The two phases of this algorithm can be clearly seen in Figures 5.39 and 5.40 as two discrete masses. The first of these expands in a heavily multiplicative manner and consists of a mixture of **ack**(n) and **nack** tokens as shown in Figure 5.40. Once the tree has been built, the **ack**(n) tokens traverse back up the *tree* rather than arbitrarily across the machine. This is why there is a gradual build-up of events followed by a steep drop-off; Figure 5.39 shows this most clearly.

The second phase requires far fewer messages because **label**(N, k_l, k_u) tokens are passed down the tree and **ack**(0) propagate back up, leading to the two smaller masses on Figures 5.39 and 5.40. A significant performance decrease in the simulator is clearly evident in Figure 5.39 resulting from the large increase in parallle traffic compared to the previous approaches.

Figures 5.41 and 5.42 show identical tree structure to that observed with the lock-step breadth-first discovery, but with different node labels. This is expected because large blocks of labels are assigned to each branch. However, the structure of the tree is the same because the simulator itself is single-threaded. Parallel tokens make the exact outcome non-deterministic in practice, but simulation enforces an artificial ordering on

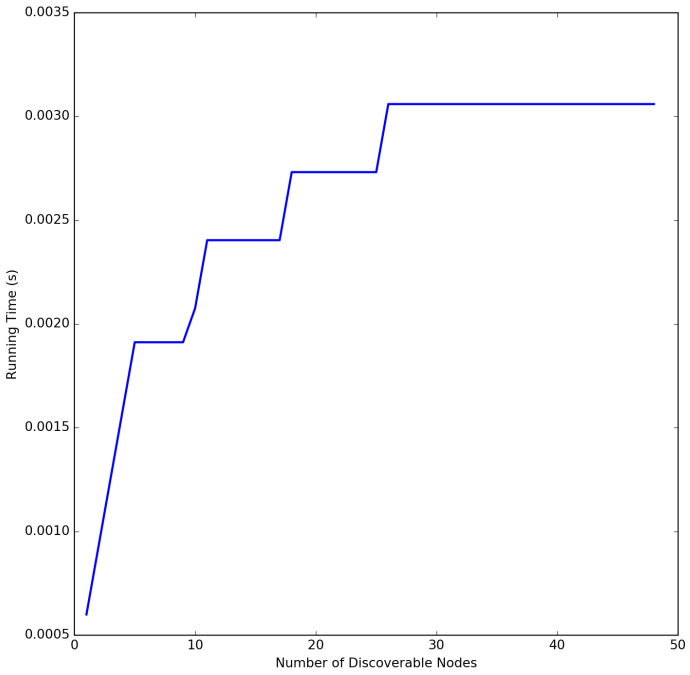


Figure 5.38: Scaling of the parallel breadth-first discovery algorithm with machine size under simulation.

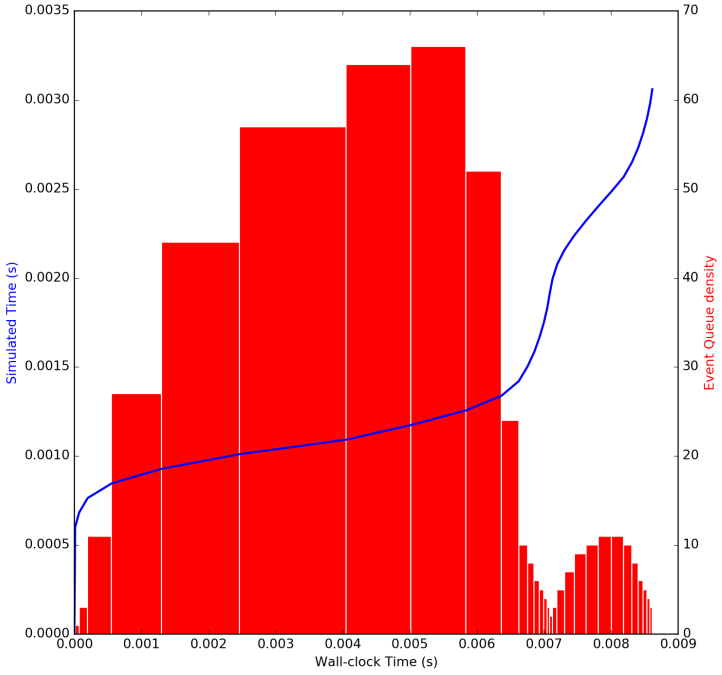


Figure 5.39: Simulated time advancement and event queue density as a function of wall-clock time for the 48 node parallel breadth-first simulation run.

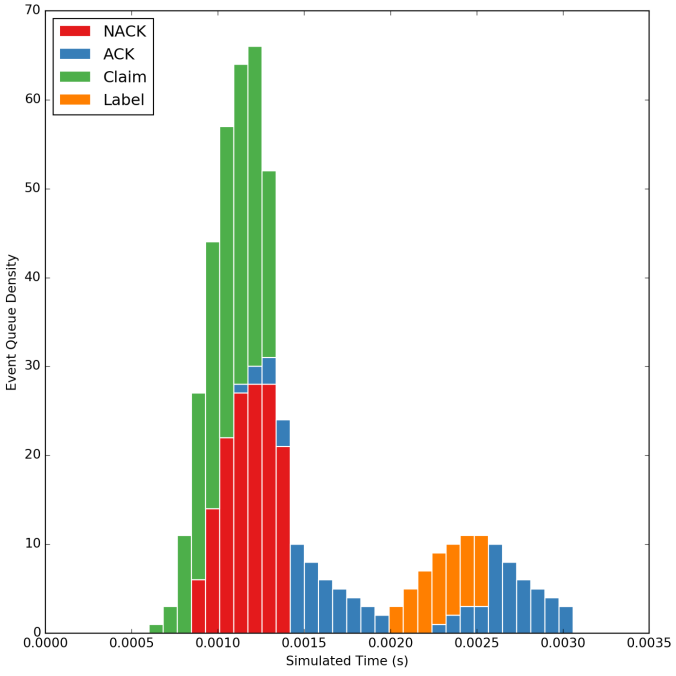


Figure 5.40: Distribution of event types (i.e., tokens) throughout the 48 node parallel breadth-first simulation run.

the events, leading to the same structure as seen before. Figure 5.42 confirms that the fault condition is tolerated, as expected.

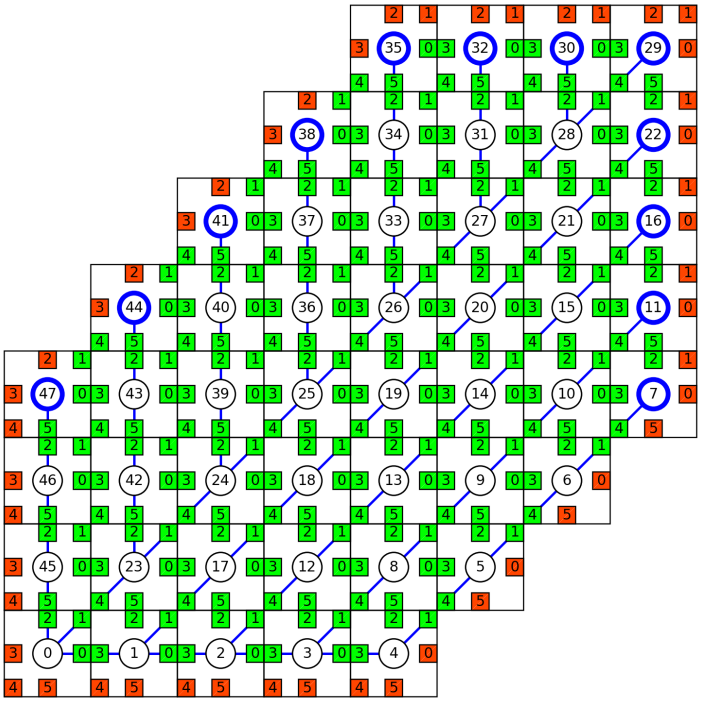


Figure 5.41: Discovered machine model state after the 48 node parallel breadth-first simulation run, showing the embedded tree and the assigned node labels.

5.5.3.2 Implementation

Figure 5.43 shows that the timings of the SpiNNaker implementation follow roughly the same stepped shape as the predicted timings, but with steeper transitions between them. The simulator is tuned to match the hardware result as closely as possible, but the single tunable delay parameter prevents the match being as close as for the previously designed algorithms. Despite this, the simulation and SpiNNaker implementation match well.

Traffic distribution across the SpiNN-103 hardware is roughly uniform, which is to be expected on a purely parallel algorithm. Unlike the other algorithms, some packets are dropped by the router due to the bursty nature of the **claim** token emission. The bar chart shows two rough bins of packet counts which correspond with the number of usable ports; edge nodes have fewer and transmit correspondingly fewer packets compared to nodes in the centre of the board.

Figures 5.45 and 5.46 show a different structure from simulation that is, overall, more balanced. SpiNNaker hardware introduces non-determinism in the network which leads to these more fairly arbitrated structures despite the token emission pattern being the

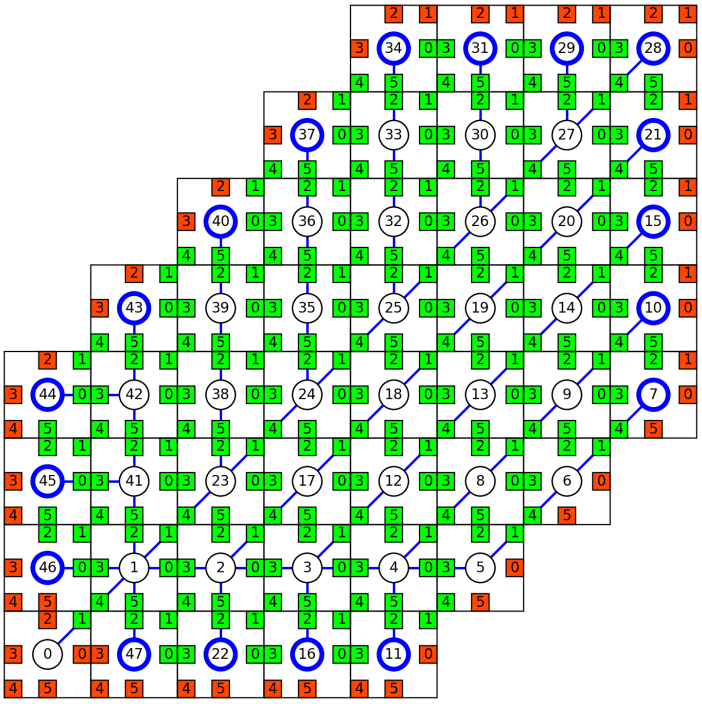


Figure 5.42: Resilience of the parallel breadth-first discovery simulation run against the fault pattern.

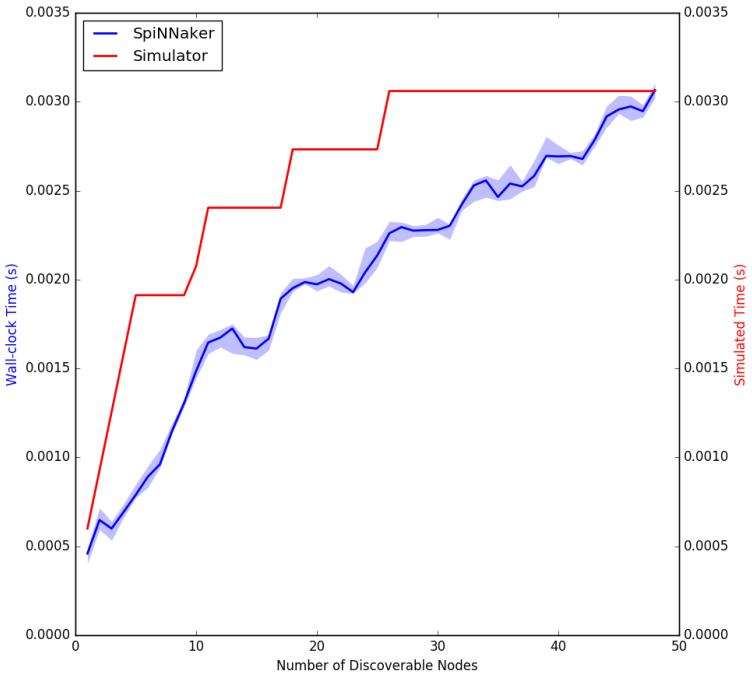


Figure 5.43: Comparison of the running time predicted by the simulator and the measured wall-clock time of 10 consecutive runs on SpiNN-103 hardware for the parallel breadth-first discovery algorithm.

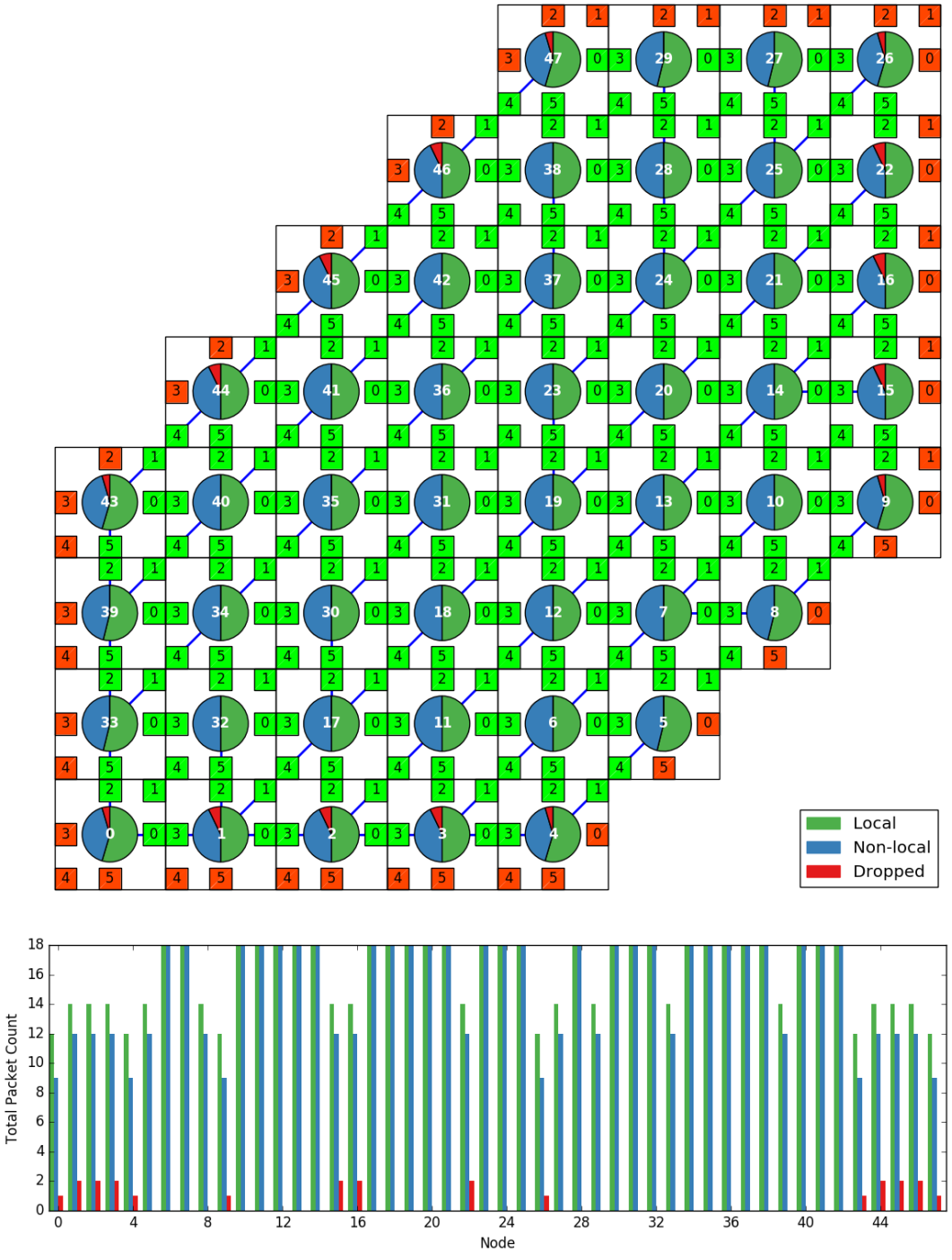


Figure 5.44: Packet distribution across the SpiNN-103 hardware for the 48 node run of the parallel breadth-first discovery run.

same as in simulation (i.e., ports with lower indices are visited first). However, the SpiNNaker hardware is not forced to deal with tokens emitted simultaneously in any particular order, unlike the simulator where multiple events for the same tick are dequeued in the order in which they were enqueued. As expected, Figure 5.46 shows that the fault pattern has again been tolerated but with a structure entirely different from simulation (Figure 5.42) because of the non-determinism of the hardware.

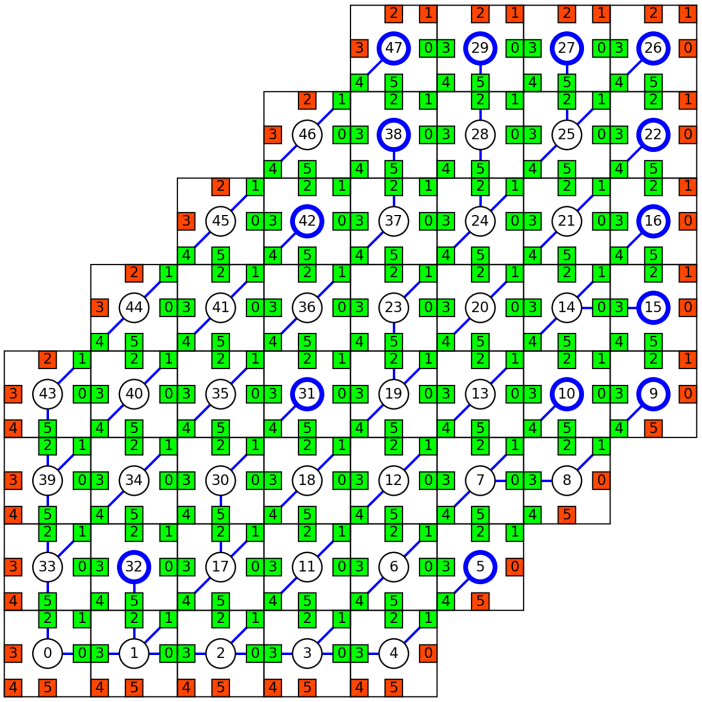


Figure 5.45: Discovered machine topology from the 48 node parallel breadth-first run on SpiNN-103 hardware.

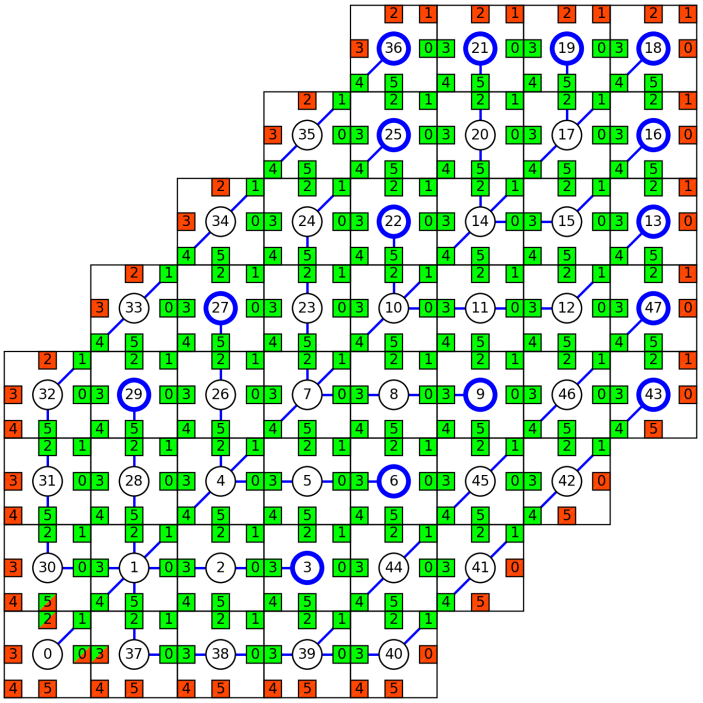


Figure 5.46: Resilience against the fault pattern of a 48 node parallel breadth-first run on SpiNN-103 hardware.

5.6 Concluding Remarks

This chapter describes a set of algorithms that survey target SpiNNaker hardware to determine the liveness of ports, uniquely label each node, and establish a control-tree that can be used to support high-level parallel programming functions. Each algorithm finishes with a barrier-like termination stage to demonstrate the usefulness of the control-tree; without it, detecting the successful completion of the algorithms would not be feasible. The algorithms have also been shown to discover faults without them preventing completion of the survey, which a addresses a potential issue that may arise on smaller SpiNNaker machines.

5.6.1 Demonstration

A SpiNN-103 board (see Figure 3.15) serves as the platform for all testing and validation of the algorithms before this point, but a real-world example would require larger and more capable SpiNNaker platforms. Figure 5.47 shows the result of lock-step breadth-first discovery of the SpiNN-104 hardware shown in Figure 3.16. Unlike the SpiNN-103 hardware, the SpiNN-104 does follow a completed torus pattern, which is indicated by the arrows bordering the visualisation.

The wavefront pattern is prominent in this figure because the root node is able to claim all six of its siblings as children, rather than just three on the SpiNN-103. Leaf nodes are again highlighted with thick blue circles which embed a jagged pattern in the figure which illuminates the two radii of the torus. The diagonal parts of the pattern highlight leaves that are on the major radius, and the straight part bisecting the figure highlights the minor radius.

Perhaps the most important observation from this figure is that there are no port malfunctions despite the scale of the SpiNN-104. Port faults were simulated whilst the algorithms were being designed over the course of this chapter, but the SpiNN-104 hardware presents the opportunity to validate against *real* faults by simply unplugging one of the inter-FPGA links shown in Figure 3.16.

Figure 5.48 shows the lock-step breadth-first survey results performed on the same SpiNN-104 hardware but with the S3-North FPGA link disconnected. The L-shaped red 'scar' in the image shows that the fault has been detected, and that the algorithm has completed successfully in spite of it. Three of the nodes failed to boot correctly, and one of them has failed to report its state despite forming part of the tree. The exact reasons for this are not clear because of the non-deterministic nature of the hardware.

5.6.2 Run-time Fault Detection

Detecting faults before the mapping is an important first step to ensuring applications function on the platform. Chapter 6 assembles the algorithms designed in this chapter into the SpinDiscover tool which uses these algorithms to build an accurate topological picture of the target machine for Loader to use for mapping purposes. However, this clearly does not detect faults that occur after an application has started running. Solving this problem falls outside the scope of the thesis, but an approach is outlined here.

SpiNNaker can be thought of as a platform capable of running multiple overlaid disjoint state-machines which change state based on packets received, and produce further packets as a consequence of state changes. The algorithms in this chapter are all implemented as state-machines of this form, but are wholly resident in each node making use of all available resources.

Without loss of generality or functionality, each node can support *multiple* state-machines that are updated simultaneously by events without necessarily directly affecting each other. As long as traffic collisions are either tolerated or avoided, these multiple state-machines can effectively behave as independent applications that are inherently time-sliced by the communications fabric.

This capability can be demonstrated using a second state machine resident in each node which consists of two states, *o_idle* and *o_halted*. Whilst in the *o_idle* state, this

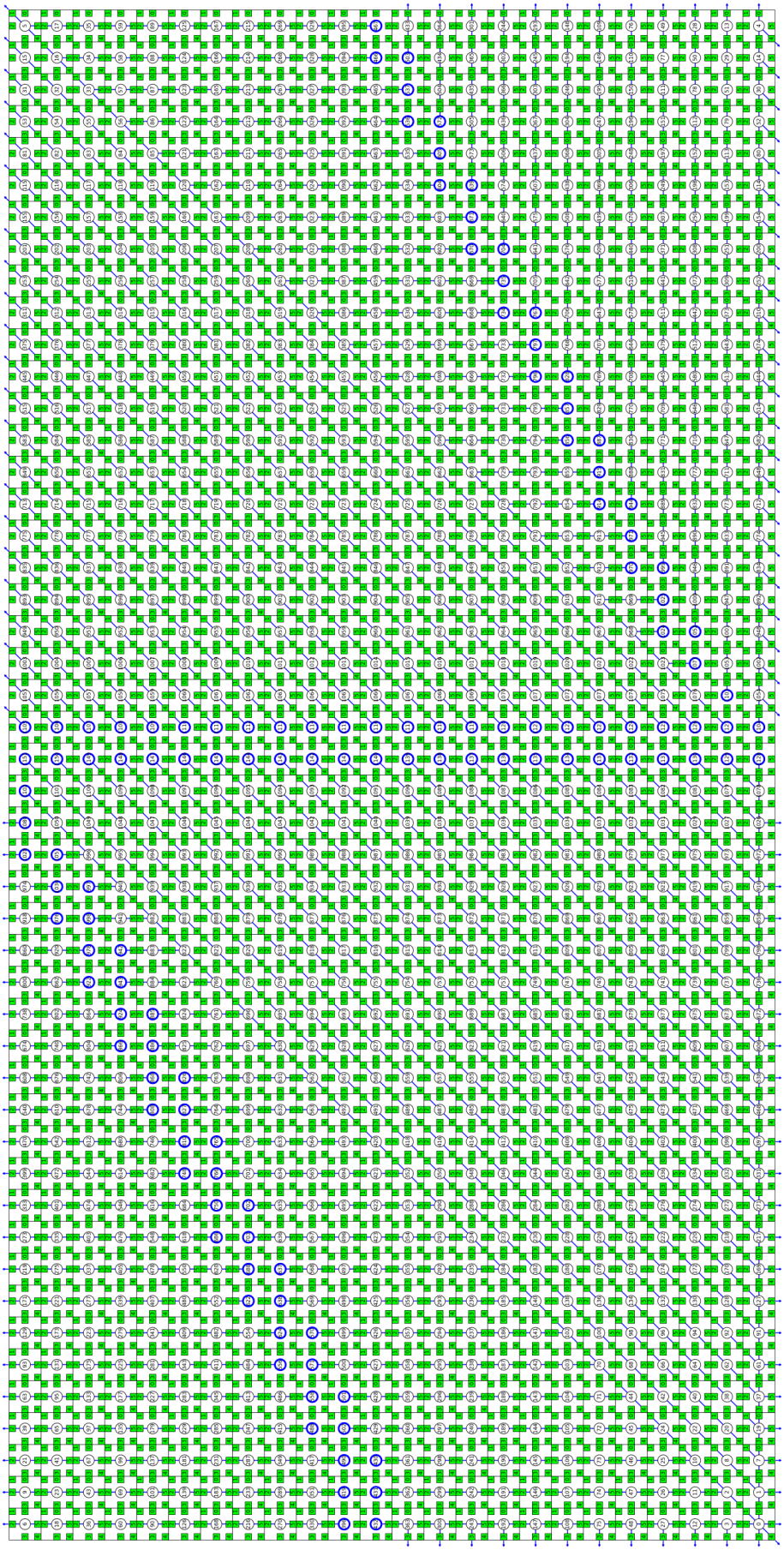


Figure 5.47: Complete survey of the SpiNN-104 hardware shown in Figure 3.16 using the lock-step breadth-first algorithm.

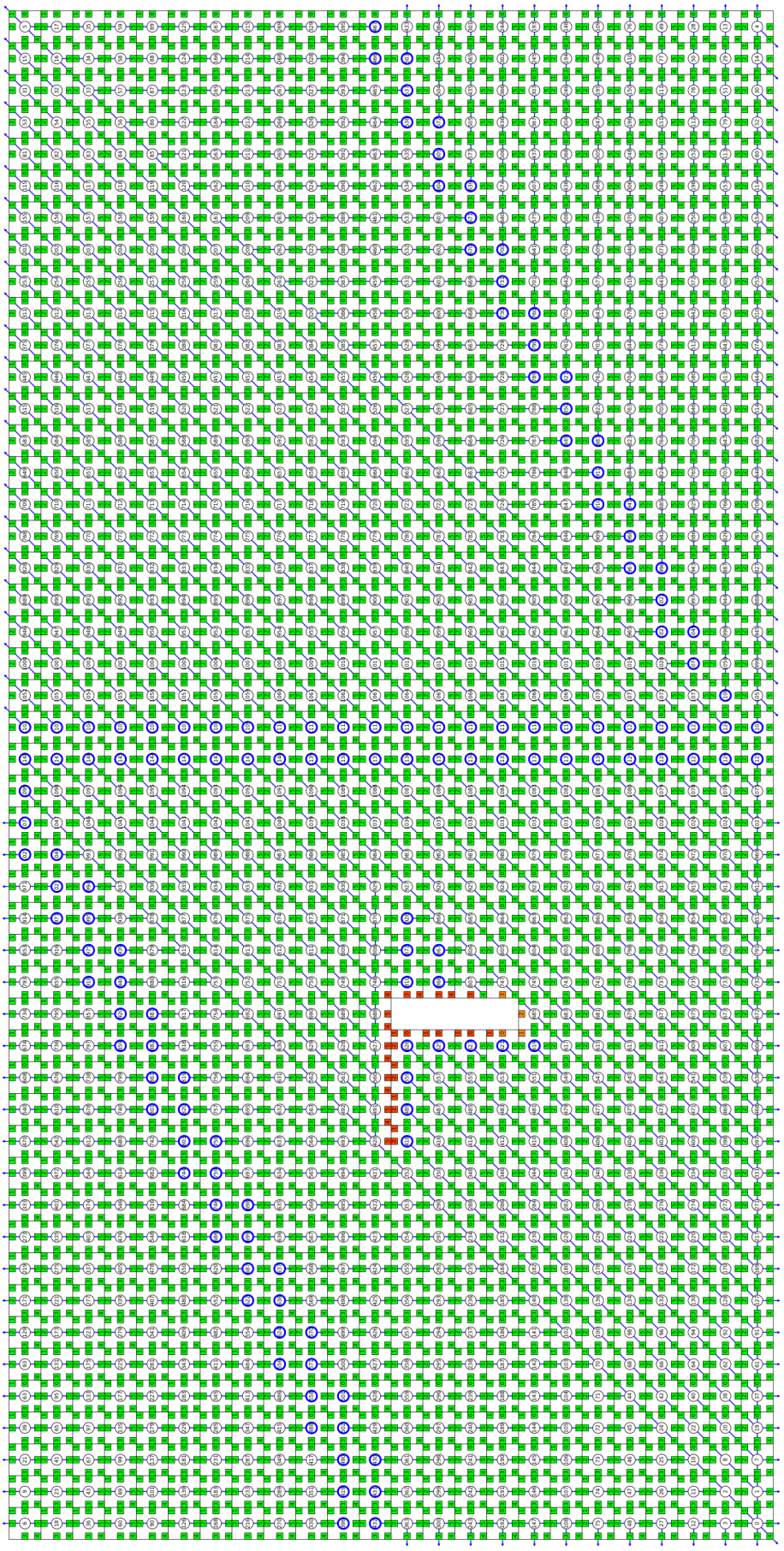


Figure 5.48: Survey of the same SpiNN-104 hardware but with a single inter-board connection unplugged.

additional state machine has no effect, but when entering the *o_halted* state it prevents *all* network messages from getting through to other applications, effectively prematurely terminating them. The **omega** token serves as the harbinger for this behaviour, causing the secondary state machine to enter the *o_halted* state upon receiving the token, and propagating the **omega** token out of all functional ports. This demonstration also shows that the state machines need not be completely isolated entities because (in this example) the secondary has an overall negative effect on the primary. Deploying disjoint state machines (or disjoint sets of interdependent state machines) permits running multiple applications in parallel without them interfering.

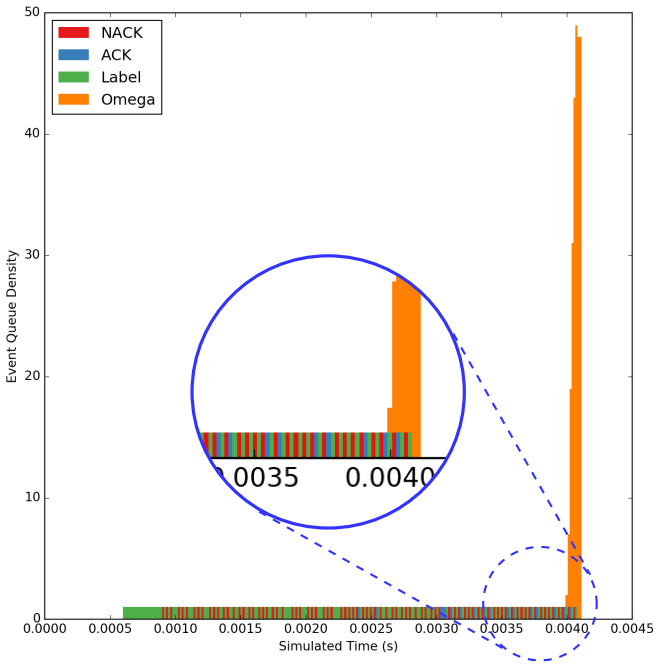


Figure 5.49: Depth-first algorithm designed in section 5.5.1 subject to an **omega** token injected at 4ms simulated time.

Figures 5.49–5.51 show **omega** being injected into the depth-first, breadth-first, and parallel breadth-first algorithms designed over the course of this chapter. In all cases, the primary algorithm continues to function whilst the **omega** is propagating through the machine model demonstrating that both state-machines can coexist. Eventually the **omega** tokens stop all activity, irrespective of the state of the primary behaviour. Figure 5.51 shows this most clearly because of the parallel nature of the simulation.

This demonstrates the concept of multiple disjoint state-machines coexisting on the same physical machine without having an effect (until the *o_halted* state takes control). Consider a state-machine embedded into SCAMP with the goal of periodically re-running the discovery algorithms, not to rebuild the control tree at run-time, but to confirm that its structure has not changed. If walking the tree determines that new faults have arisen,

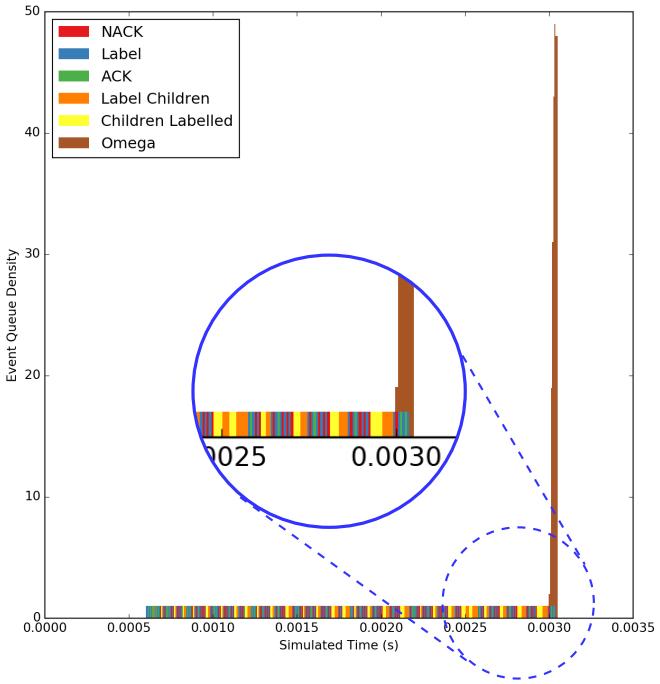


Figure 5.50: Breadth-first algorithm designed in section 5.5.2 subject to an **omega** token injection at 3ms simulated time.

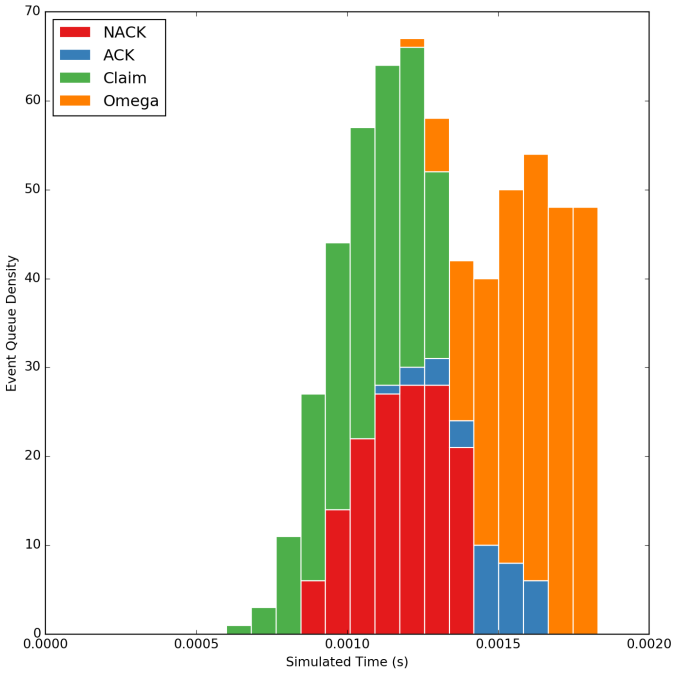


Figure 5.51: Parallel breadth-first algorithm designed in section 5.5.3 subject to an **omega** token injection after about 1ms simulated time.

then appropriate measures can be taken to deal with them. In the simplest case, this would be reporting the error to the outside world so that applications can be remapped and rerun, but the ideal case (and an interesting avenue of further work) would be for the application to be migrated around the fault without any significant impact on the overall behaviour.

5.6.3 Review

To conclude, this chapter began with the problem statement identifying three issues that must be addressed:

1. Hardware faults must be discovered before any problem mapping takes place,
2. A complete model of the current state of the hardware must be assembled from the discovered structure of the machine, and
3. A control tree must be embedded to support common parallel programming constructs.

These points have been addressed and the solutions verified both in simulation and on physical SpiNNaker hardware. Port faults are discovered on the physical hardware using the $\alpha - ping$ designed in section 5.4.2, section 5.5 then presented a set of algorithms that can discover the structure of a machine whilst simultaneously constructing a control tree. All of the visualisations show that a model of the machine can be reconstructed based on the discovery information.

In the next chapter, these results are augmented with configuration algorithms to form a complete pre-mapping tool-flow that ultimately leads to a demonstrative non-neural application being built and tested.

Chapter 6

Non-neural Application

6.1 Design

Solving complex equations numerically is a common use-case for parallel computation. The problem-space is typically discretised into some kind of mesh where the partitions are handled individually as shown earlier in Figure 2.5. Care must be taken at the boundaries of these sub-spaces to ensure numerical accuracy.

Discretisation of problem spaces in this manner naturally forms a sparsely connected graph representing how each of the discrete points affect one another; usually, but not always, these connections are nearest-neighbour. Recall from Chapter 3 that problem graphs are already of this form, therefore SpiNNaker allows problem spaces to be discretised to the point were (potentially) each element of the discrete problem space can be assigned to an *individual processor* [1].

This chapter demonstrates this concept by designing an application to determine the steady-state result for the canonical heat diffusion equation,

$$\frac{\partial T}{\partial t} - \alpha \nabla^2 T = 0, \tag{6.1}$$

in two dimensions. Specific material properties are not considered because the intention is for this application to demonstrate a capability, rather than to compute a physically significant result. Therefore, α of equation (6.1) can be tuned to a value appropriate for the simulation even if this would require material characteristics that would be impossible in nature.

$$T(0,x,y) = \begin{cases} 0° & (x,y) = (X,Y), \\ 255° & x = y = 0, \\ 127° & \text{otherwise.} \end{cases} \tag{6.2}$$

The width and height of the plane upon which equation (6.1) is to be solved are given as X and Y, respectively, where $X = Y = 1\text{m}$. Equation (6.2) states the initial conditions that will be applied to equation (6.1), and equations (6.3) and (6.4) state the boundary conditions. The simulation contains exactly one heat source at $(0,0)$ which holds a constant temperature of 255°, and one heat sink at (X,Y) which is fixed at 0°.

$$\frac{\partial}{\partial y}T(t,x,0) = 0 = \frac{\partial}{\partial y}T(t,x,Y), \qquad x \in [0,X]. \tag{6.3}$$

$$\frac{\partial}{\partial x}T(t,0,y) = 0 = \frac{\partial}{\partial x}T(t,X,y), \qquad y \in [0,Y]. \tag{6.4}$$

Figure 6.1 shows the discrete representation of the plane over which equation (6.1) will be solved. The blue vertices form the problem graph that ultimately describes the application, and the orange vertices represent constant-values (clamp vertices) that ensure the mathematics remain valid. Two grid sizes of 20×20 and 90×90 are used for the simulations, where the α parameter is tuned appropriately in each case.

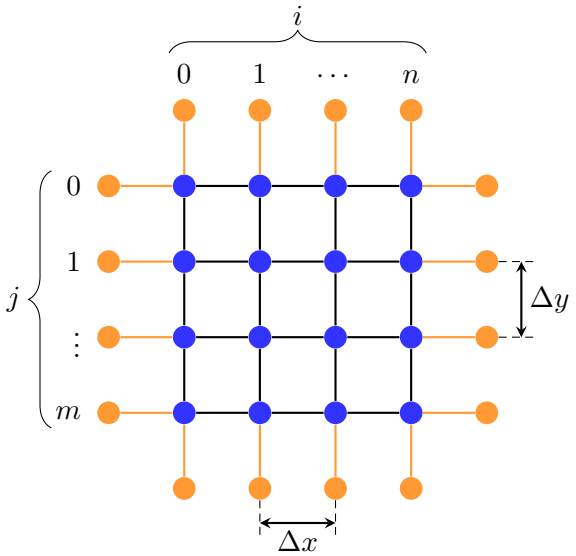


Figure 6.1: Discretised problem grid showing problem nodes in blue and clamp nodes in orange.

Each blue vertex in Figure 6.1 must compute its temperature using the discrete form of equation (6.1). Spatial separation between the vertices is given as Δx and Δy which are uniform in their respective dimensions. The exact value at any point on this grid is given by equation (6.5).

$$\frac{\partial}{\partial t}T(t_k, x_i, y_j) = \alpha \left[\frac{\partial^2}{\partial x^2}T(t_k, x_i, y_j) + \frac{\partial^2}{\partial y^2}T(t_k, x_i, y_j) \right], \tag{6.5}$$

where $x_i = i \cdot \Delta x$, $y_j = j \cdot \Delta y$, and $t_k = k \cdot \Delta t$. Careful manipulation of the Taylor Series expansions around t_k, x_i, and y_j show that [141]:

$$\frac{\partial T}{\partial t} \approx \frac{T(t_{k+1}, x_i, y_j) - T(t_k, x_i, y_j)}{\Delta t} \tag{6.6}$$

$$\frac{\partial^2 T}{\partial x^2} \approx \frac{T(t_k, x_{i-1}, y_j) - 2T(t_k, x_i, y_j) + T(t_k, x_{i+1}, y_j)}{(\Delta x)^2} \tag{6.7}$$

$$\frac{\partial^2 T}{\partial y^2} \approx \frac{T(t_k, x_i, y_{j-1}) - 2T(t_k, x_i, y_j) + T(t_k, x_i, y_{j+1})}{(\Delta y)^2} \tag{6.8}$$

Equations (6.6)–(6.8) can be substituted into equation (6.5) to yield the complete discrete form. The notation $T^k_{i,j} = T(t_k, x_i, y_j)$ is used here for brevity and clarity:

$$\frac{T^{k+1}_{i,j} - T^k_{i,j}}{\Delta t} = \alpha \left[\frac{T^k_{i-1,j} - 2T^k_{i,j} + T^k_{i+1,j}}{(\Delta x)^2} + \frac{T^k_{i,j-1} - 2T^k_{i,j} + T^k_{i,j+1}}{(\Delta y)^2} \right] \tag{6.9}$$

Equation (6.9) is the *difference equation* equivalent of Equation (6.1) which clearly depends on the temperature (i.e., state) of the current vertex and its neighbours. Solving the equation iteratively requires that each vertex computes a new value for $T^{k+1}_{i,j}$ and transmits it to its neighbours. Each vertex must therefore compute:

$$T^{k+1}_{i,j} = T^k_{i,j} + \alpha \cdot \Delta t \left[\frac{T^k_{i-1,j} - 2T^k_{i,j} + T^k_{i+1,j}}{(\Delta x)^2} + \frac{T^k_{i,j-1} - 2T^k_{i,j} + T^k_{i,j+1}}{(\Delta y)^2} \right]. \tag{6.10}$$

Using equal spacing in both dimensions of the plane (i.e., $\Delta x = \Delta y$) allows this equation to be written as:

$$T^{k+1}_{i,j} = \beta \left[T^k_{i-1,j} + T^k_{i+1,j} + T^k_{i,j-1} + T^k_{i,j+1} \right] + T^k_{i,j}(1 - 4\beta), \qquad \beta = \alpha \frac{\Delta t}{(\Delta x)^2} \tag{6.11}$$

It can be shown that for equation (6.11) to be numerically stable and converge, the following criterion must hold [141]:

$$\beta \leq \frac{1}{4} \tag{6.12}$$

Recall from Chapter 4 that an application contains *handlers* that respond to packet arrivals and the regular timer tick. The IntHand API (introduced in Figure 4.1) installs both of these handlers so that it can search for the appropriate edge and device (vertex) parameters, before invoking callbacks (registered by the application) to handle the specific behaviours. *Edge handlers* are invoked after this look-up is triggered by a packet arrival, and *device handlers* are invoked for each device after the timer tick fires.

Problem graph specifications include *parameters* (which are immutable) and *state variables* which are mutable. The discrete plane upon which equation (6.11) is solved has uniform thermal impedance (Figure 6.1), hence there are no edge parameters required. Inspection of equation (6.11) shows that the device handler must know the temperature values for its neighbours and β. Therefore, the device specification must include β as a parameter, and state variables to hold the neighbour temperatures (known as *ghosts*) and the current temperature of the device itself.

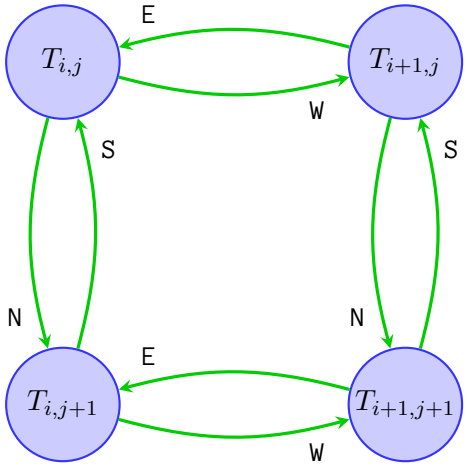


Figure 6.2: Problem graph fragment based on the discrete mesh shown in Figure 6.1, highlighting the edge types for storing neighbouring temperature values.

Ghosts must only be updated using values from the appropriate neighbour device for the simulation to have any physical meaning. Both devices and edges have a type defined against them in the problem graph specification, which are used to address this issue. In equation (6.11), the exact identifiers of the neighbouring devices are not important, only their relative location is. Figure 6.2 shows a fragment of the problem graph that describes Figure 6.1, with four edge types shown as compass directions. These are used to ensure that updates from (for example) the northern neighbour are only ever written to the northern ghost; illustrated by the EDGEHANDLER procedure of Algorithm 6.1.

In the DEVICEHANDLER procedure of Algorithm 6.1, T_{thresh} is introduced as the smallest upper bound on ΔT; once it falls below T_{thresh}, the device has converged. Local convergence implies nothing of the global convergence of the problem, but it is used to inhibit the MC emission of new temperature values; thus when *all* problem devices have

Algorithm 6.1 Heat diffusion implementation.

Require: `edge.type` $\in$ {`N`, `E`, `S`, `W`}

```
1:  procedure EDGEHANDLER(T)
2:      vertex.ghost[edge.type] ← T                ▷ edge.type set in problem graph.
3:  end procedure

4:  procedure DEVICEHANDLER
5:      T' ← β · Σ_{T∈vertex.ghost} T + (1 − 4β) · vertex.value    ▷ Solve equation (6.11).

6:      if ⌊vertex.value − T'⌋ < T_thresh then
7:          EMIT(T')                               ▷ MC broadcast to all neighbours.
8:      end if

9:      vertex.value ← T'
10: end procedure
```

converged locally, the problem has globally converged. The floor function on line 6 is implicit because of the unsigned 8 bit integer arithmetic used to solve the equation.

As devices are updated when the timer tick fires, tuning the timer period is essential to maintaining biological fidelity in neural applications. This application, however, is obviously not real-time, and the timer tick is re-purposed to throttle the number of packets being injected into the communications network. If all problem devices emitted new values without a rate-limit, the network would be overwhelmed and there would no longer be any guarantee that the application would converge at all.

The timer tick also implies that the application will solve the equations at each vertex using the Jacobi update method [141, pg. 237], where the new value, $T_{i,j}^{k+1}$, is computed once the values from the current time-step are available. However, the non-determinism of the SpiNNaker network suggests that some update messages will be lost, causing the computation to be performed on old data. This is akin to the Gauss-Seidel update method [141, pg. 238] which permits the use of the most up-to-date value. Usually this justifies the usage of values calculated in the current time-step to be used in the current time-step, but in this case it permits the use of older data without preventing convergence.

6.2 Simulation

As with the algorithms in Chapter 5, the application is first validated using a simulator before moving it to SpiNNaker. Figure 6.3 highlights the part of the tool-flow used for this. Answer was introduced in Chapter 4 as a single-threaded event-driven simulator which provides a SpiNNaker-like environment. Code written for this environment is

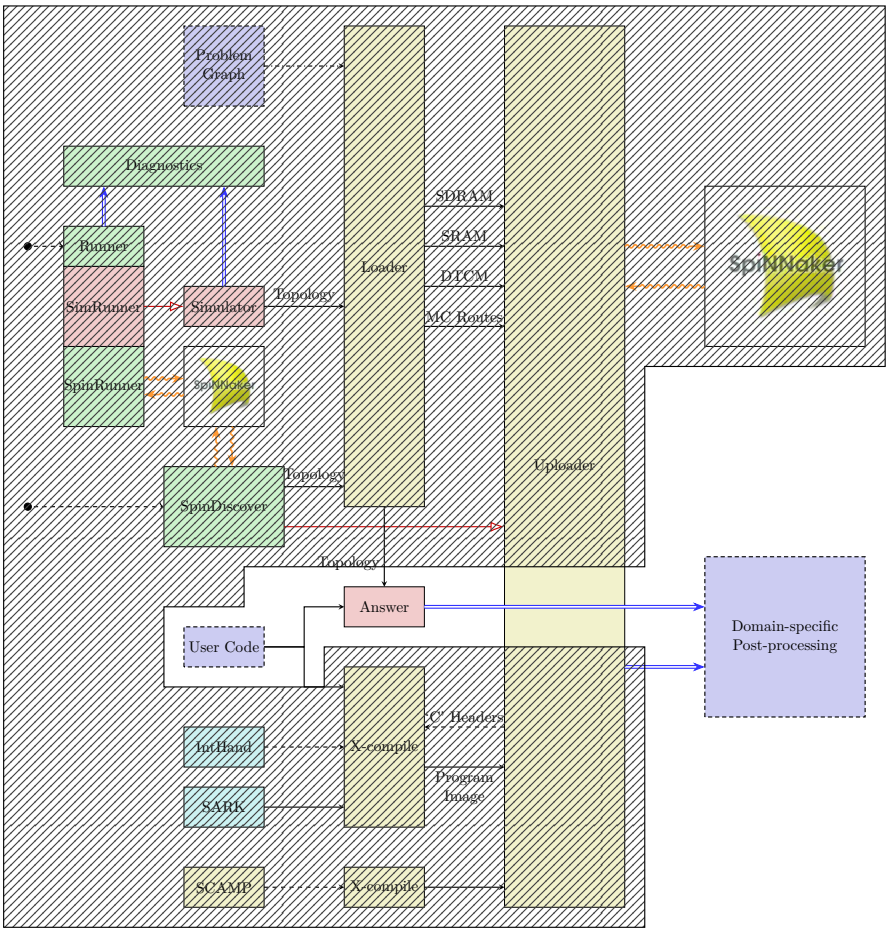
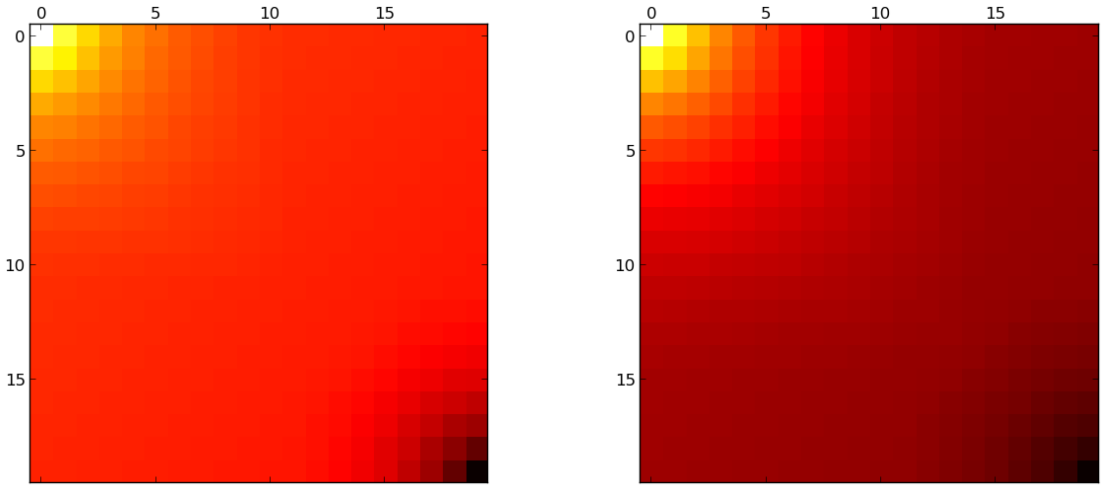


Figure 6.3: Stages of the toolflow shown in Figure 4.1 used to verify the function of the heat diffusion application.



(a) Numerically important but physically unimportant intermediate result.

(b) Steady-state solution.

Figure 6.4: Answer simulation result from a 20×20 grid.

already of the correct form to be compiled for SpiNNaker, unlike the SimRunner block which is purely behavioural.

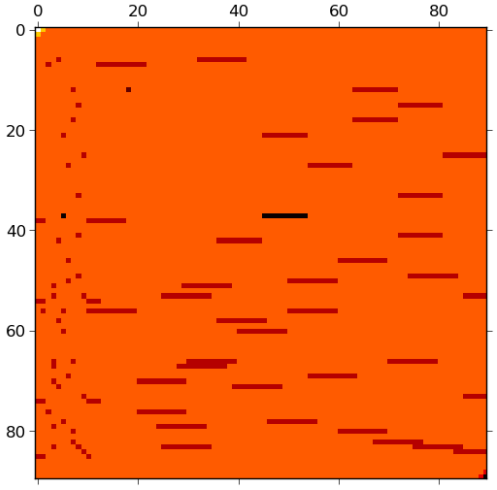
Figure 6.4 shows the problem device temperatures for a problem graph composed of the fragments shown in Figure 6.2. Each sub-figure shows the temperatures for different values of k: Figure 6.4a shows an intermediate step which is numerically significant because it forms part of the trajectory of the computation but has no physical meaning yet; Figure 6.4b shows the steady-state solution which is both numerically significant and physically correct.

The complete problem graph is not a homogeneous composition of Figure 6.2 fragments; if it were, the simulation would ultimately converge to constant value without showing a gradient. In general, a device solves equation (6.11), but devices $T_{0,0}$ and $T_{n,m}$ (in Figure 6.4) are the *source* and *sink* devices respectively. Instead of computing an updated value for their temperature, they emit the constant value assigned to them in the problem graph. $T_{0,0} = 255$ and $T_{n,m} = 0$, thus leading to the gradients shown in Figure 6.4. SpiNNaker does not have any cores containing floating-point units, so the simulation instead uses 8 bit unsigned integer arithmetic.

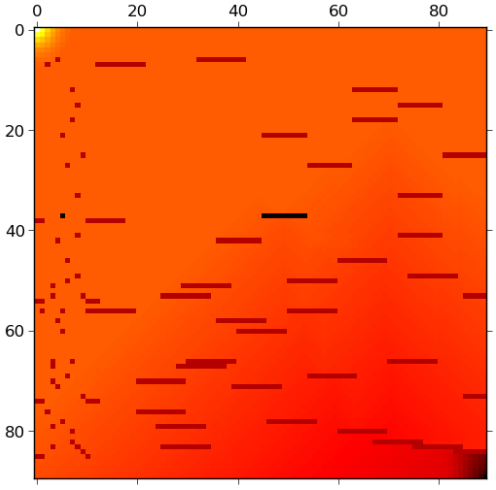
It is clear from Figure 6.4b that the problem graph method of solving equation (6.11) converges for sufficiently large values of k. However, Answer does not simulate the non-determinism of the SpiNNaker network. Disabling a pseudo-random assortment of cores hosting the devices emulates this condition, because there is essentially no difference (discernible by the receiving problem device) between temperatures being lost and temperatures having never been sent.

Figure 6.5 shows the trajectory of an Answer simulation on a 90×90 problem device grid with the same constant value devices as before (i.e., $T_{0,0} = 255$ and $T_{n,m} = 0$), but with 5% of the worker cores disabled. The cores that have been disabled are most clearly seen on Figure 6.5a as the darker problem devices in the grid. Figures 6.5a–6.5c are each numerically significant but, as before, are not yet physically valid. Figure 6.5d shows the steady-state solution which is globally correct despite the significant number of faults introduced.

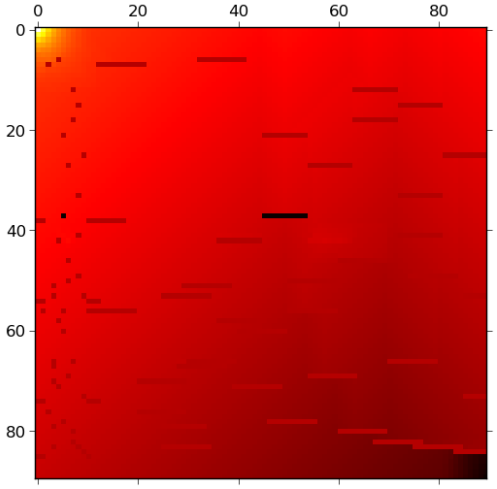
Specific initial conditions are not important, but it is clear from equation (6.11) that $T_{i,j}$ cannot be uniformly zero across the entire grid. Algorithm 6.1 reacts purely to updates from neighbours, so there must exist *at least one* initial message. A useful property of both Algorithm 6.1 and equation (6.11) is that any update that will cause a change in temperature will begin the simulation. The problem devices of both Figure 6.4 and Figure 6.5 are given random initial temperatures, with the clamp devices, $T_{0,0}$ and $T_{n,m}$, emitting their temperatures immediately to begin the packet storm.



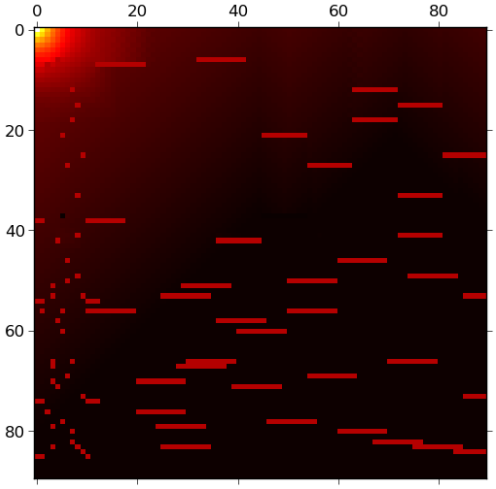
(a) $k = 1$: first time-step after initial conditions. Devices mapped onto disabled cores are clearly visible.



(b) $k = 100$: devices near the source and sink devices are beginning to take the correct trajectory.



(c) $k = 200$: gradient more pronounced despite the significant number of faults introduced.



(d) Steady-state solution which is globally correct, despite the faults introduced.

Figure 6.5: Answer simulation result from a 90×90 grid with 5% of the simulated cores disabled.

6.3 Implementation

Figure 6.6 shows the complete tool-flow for running the heat diffusion application (and indeed any appropriate application) on SpiNNaker. The function of the user code has been proved by Answer, and the capability of the machine discovery algorithms has been shown in Chapter 5. To complete the flow, SpinDiscover must provide Loader with an accurate model of the physical hardware so that the problem graph and user code can mapped correctly.

SpinDiscover packages the algorithms described in Chapter 5 with the two described in this section that configure the P2P router and ensure that nodes have a record of *all*

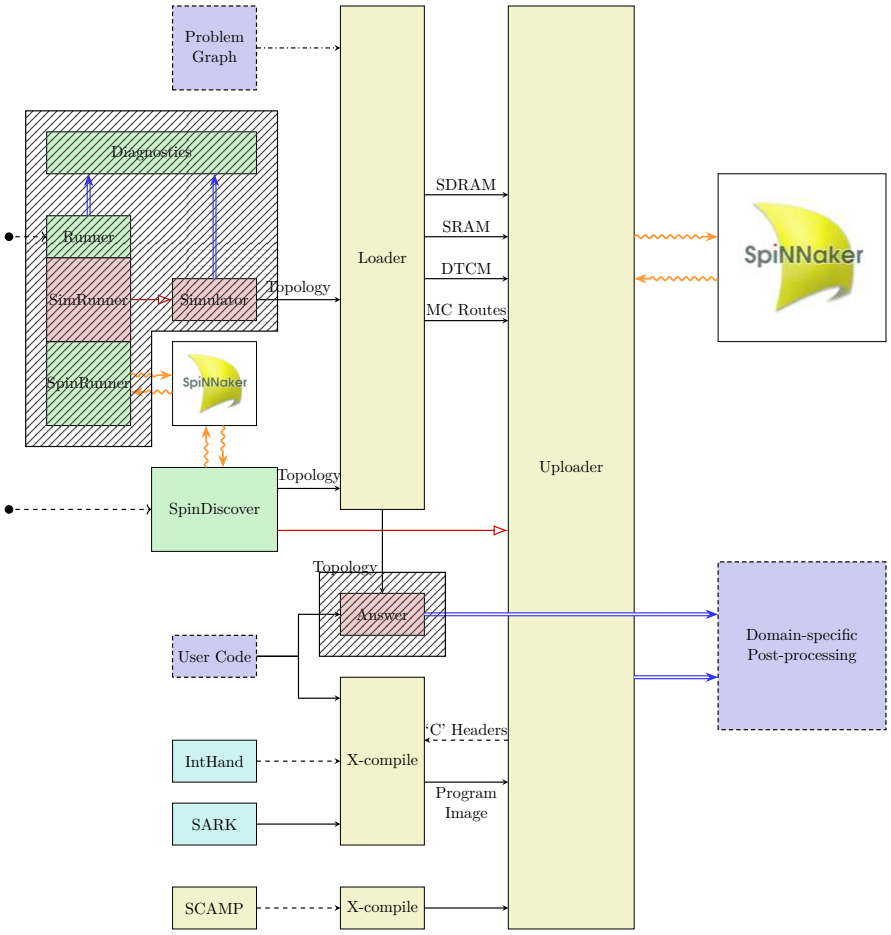


Figure 6.6: Stages of the toolflow shown in Figure 4.1 used to execute the heat diffusion application on SpiNNaker and collect the results.

of their neighbours to complete the topological description of the system. Once they have been executed on the target SpiNNaker machine, a complete topological map is downloaded by SpinDiscover and written into a set of files for the Loader.

6.3.1 P2P Table Construction

P2P routing via the P2P routing tables has been described in Chapters 3 and 4 and shown visually in Figures 3.20, 3.21 and 4.3. The Loader uses established P2P routes to route edges between problem devices that have been allocated to nodes separated by more than a single hop. The Uploader uses P2P messages to perform directed loads of heterogeneous software and (far more commonly) the data-structures produced by the Loader.

Establishing the P2P tables can only occur after a discovery algorithm from Chapter 5 has traversed the machine, labelling the nodes and establishing a control tree. The tree provides a simple method of completion detection, allowing the algorithm to operate in a highly parallel manner using the following tokens:

- **label**(l): Sent to a node to apprise it of a route.
- **ack**: Reported to a parent node from a child node when the P2P table is complete populated. This is determined by simple comparing the length of the table to the total number of nodes discovered in the system.

Like the discovery algorithms, it is implemented as a state-machine:

- *idle*: node is waiting for the process to be started. Upon receiving the first **label**(l), the node broadcasts its own label from all bidirectional ports and then enters the *building* state.
- *building*: all **label**(l) tokens received (including the one mentioned above that causes the state transition) are used to populate the P2P table. Entry l in the P2P table is looked up If it has not yet been set, the receiving port is stored against it, a local counter, c, is incremented, and l is rebroadcast out of all bidirectional ports *except* the one through which it arrived. Conversely, if the entry was already set, then no further action is taken for this token.
- *terminal*: when $c = N$ (N being the total number of nodes discovered, set by any one of the algorithms in Chapter 5) and a **ack** has been received from *all child nodes*, the node automatically enters this state to inhibit the processing of any further tokens and issues a **ack** to its parent. Leaf nodes will report immediately when $c = N$ because there are no child nodes to report.

Figure 6.7 shows the state transition diagram. Once the root node enters the *terminal* state, it issues an SCP message to the host computer running SpinDiscover which prompts it to do a directed download of the P2P tables. Contiguous labels reduce the size of the data-structure because the index of an entry implies its location in the table. Contrast Figures 3.20 and 3.21 with Figure 4.3. The former requires only a byte per entry in the downloaded state because the labels form a contiguous and complete sequence. In the latter, there is no sequence, hence each entry must be a tuple of target identifier and the appropriate port.

Figure 6.8 shows four P2P entries for every node on the SpiNN-103 system configured using this algorithm. The process itself is non-deterministic because the the volume of traffic, which is reflected in the routes that have emerged. Each node in Figure 6.8 is highlighting the port used to route P2P messages to the node highlighted by the thick blue circle. Using Figure 6.8b as an example: node 27 sends *all* P2P traffic destined for node 15 out of its southern (5) port; it does not (nor needs to) know to complete route because that is handled by node 20, the recipient of the initial message.

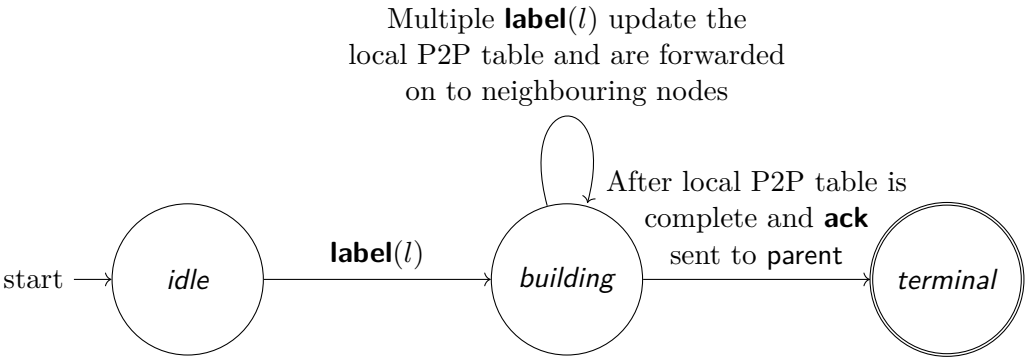
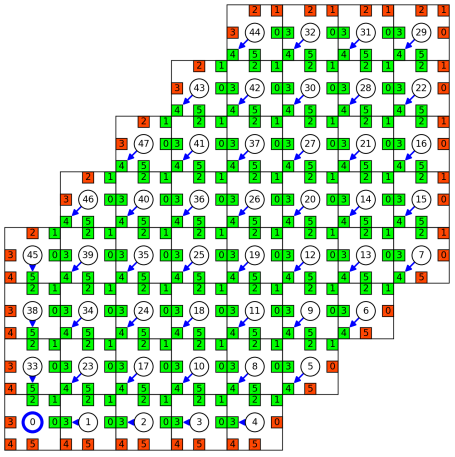
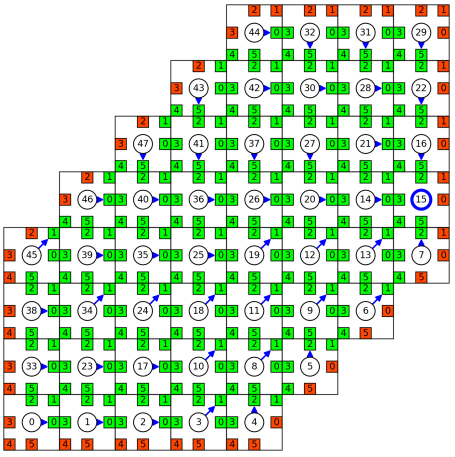


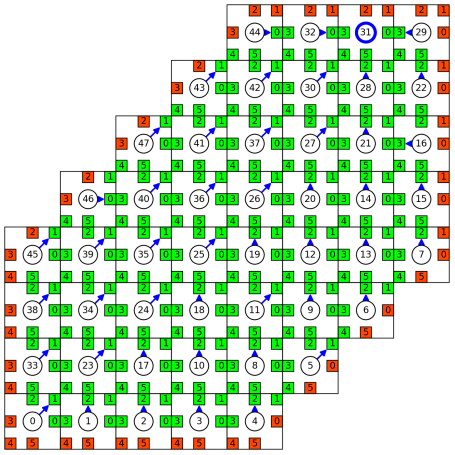
Figure 6.7: State transition diagram for the P2P table construction algorithm.



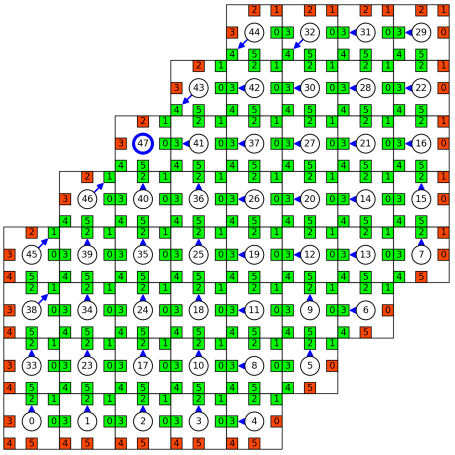
(a) All routes to node 0.



(b) All routes to node 15.



(c) All routes to node 31.



(d) All routes to node 47.

Figure 6.8: Select slices through the P2P tables of all nodes in the SpiNN-103 system. Each node shows the port it uses to communicate with the node highlighted with the thick blue circle.

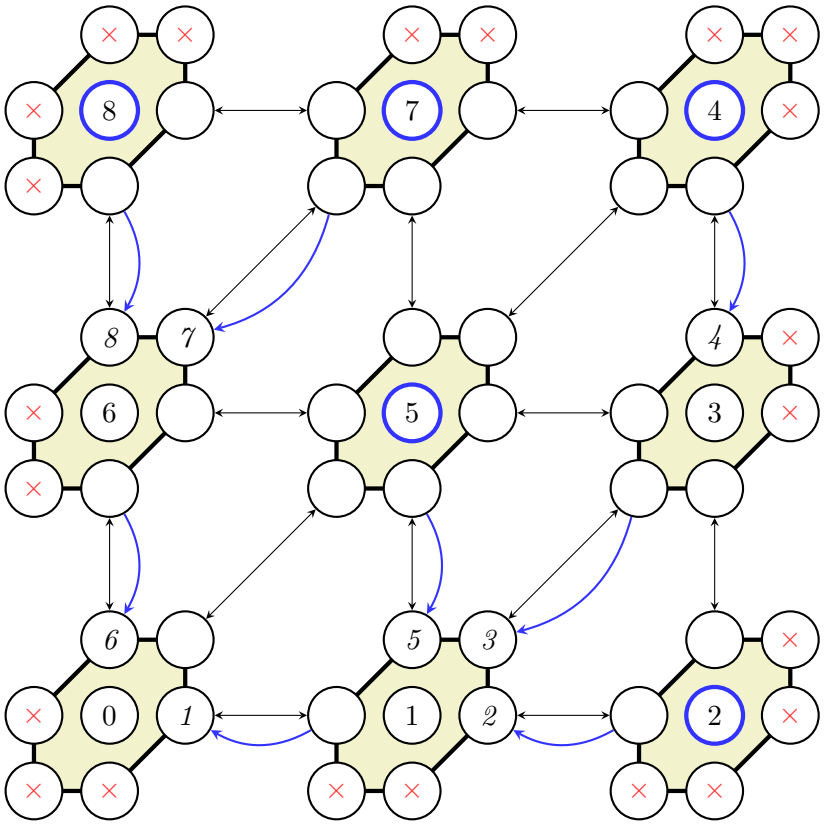


Figure 6.9: A 3×3 SpiNNaker mesh labelled by the parallel breadth-first discovery algorithm showing which neighbour identifiers are known.

6.3.2 Completing the Connectivity Model

Figure 6.9 shows the state of a small SpiNNaker system after parallel breadth-first discovery (section 5.5.3). The blue arrows show the control tree hierarchy and the blue circles highlight the leaf nodes. As the algorithm (or indeed any of the discovery algorithms designed in Chapter 5) progresses, nodes label their ports locally according to the labels their children will assign themselves. However, as can be seen in Figure 6.9, this leaves a large number of ports unlabelled, including that of the parent. Leaf nodes have no knowledge of any neighbouring node.

Loader assumes the description of the machine is purely topological. In principle, it could applied to a system that merely *behaves like* SpiNNaker. For Loader (or any other topology-oriented tool) to construct an accurate model of the specific SpiNNaker system, the blanks shown by Figure 6.9 must be filled.

The "continuity algorithm" uses the following tokens:

- **visit**(l): Passed to child nodes to inform them of the identifier of their parent, l, and also to enter the *mapping* state.

- **apprise**(l): Carries the label, l, of a node in the *mapping* state to all neighbours connected by bidirectional ports.
- **terminal**: Sent from child to parent nodes when all ports have been labelled.

And the following states:

- *idle*: Node is waiting to be visited by its parent.
- *mapping*: Entered after receiving a **visit**(l) token from its parent node. l is stored against the parent port, and the local identifier, l', is broadcast to all bidirectional ports that are neither children nor the parent via an **apprise**(l') token. A corresponding **visit**(l') is sent to all child ports. Once all ports have been labelled and **terminal** tokens have been received from all child ports, the node automatically enters the *terminal* state and issues its parent with **terminal**.
- *terminal*: All ports have been labelled and all children also have labelled ports. Leaf nodes enter this state immediately when ports are labelled because they have no child nodes.

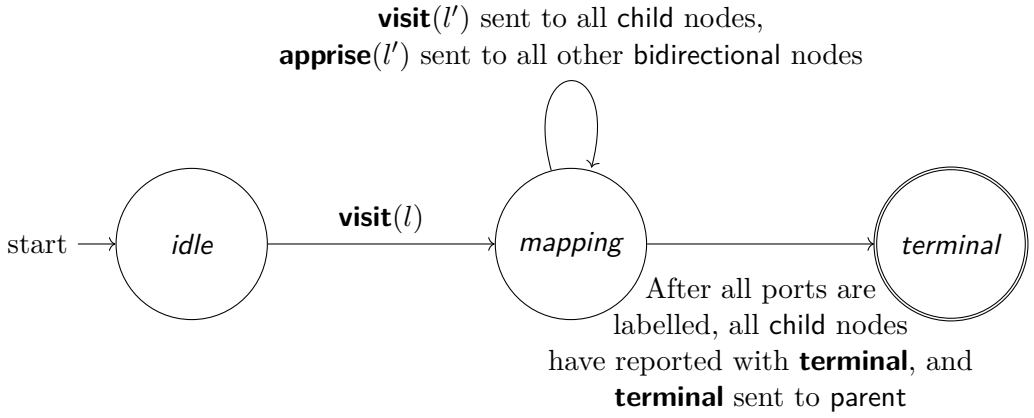


Figure 6.10: State transition diagram for the continuity algorithm.

Figure 6.10 shows the state transition diagram. Similarly to the P2P configuration algorithm explained in the previous section, the root node issues an SCP message to the host once it enters the *terminal* state. This signals SpinDiscover to query each node for the port labels and the number of functioning worker cores using specific SCP commands. Once complete, the information is assembled into file which is passed to the Loader (shown by the "Topology" arrow pointing from SpinDiscover to the Loader in Figure 6.6).

6.3.3 Reducing Packet Flux—The "Ping-pong" buffer

Biological neurons fire at a rate of roughly 10Hz (sometimes as high as 100Hz in specialised sub-systems), and the SpiNNaker connection fabric has been designed to support large fan-outs operating at this rate [35]. The timer tick period is usually set to about 1ms for most neural applications, to achieve biological real-time updates without the models becoming numerically unstable. A consequence of this is that devices (neurons) may be updated 10's or 100's of times before a message is emitted (i.e., a spike is fired).

In this application, *every* problem device (vertex) can emit a new temperature message on *every* timer tick (see Algorithm 6.1), which produces a traffic pattern that falls outside of the original design intent. Increasing the timer tick period does not counteract the issue because the instantaneous volume of packets being emitted remains the same irrespective of the device update frequency. To make matters worse, the packet arrival handler is essentially a blocking operation as the core-local communications controller cannot accept new packets until the receipt of the current one is acknowledged by clearing the interrupt flag. There are small buffers between the router and the cores that absorb some of the back-pressure that this applies, but they are not sufficient to tolerate the volume of traffic in this application. Therefore, the packet arrival handler must be **fast**.

As mentioned in section 6.1, IntHand uses the packet arrival handler to search the data-structures generated by Loader for the appropriate parameters corresponding to the incoming edge. Recall that applications use MC messages to communicate between problem devices (Chapter 4), and that MC packets carry both a source address (the *key*) and an optional payload. The keys exist in a 32 bit space, hence the parameter look-up cannot simply be an index. A *key-tree* forms part of the Loader-generated data-structures, enabling the look-up to be a binary search which completes fast enough (despite the SDRAM latency) for applications with biological packet emission rates. However, the traffic volume in this application falls outside of anything that resembles a biologically-realistic traffic volume, and even this binary search is too slow.

The solution is to defer the parameter look-up to the timer tick handler, and reduce the packet arrival handler to simply buffering all packets that arrive. Care is required here because the interrupts in SpiNNaker have associated priority levels such that lower priority handlers can be interrupted by those with a higher priority. The timer tick is typically a slow operation because it is used to update all of the problem devices on a node, which can be as many as 10^3, and hence is assigned a low priority. For the reasons described earlier, the packet arrival interrupt must be handled quickly and hence has a high priority. This bears resemblance to two threads sharing a common resource. Section 2.1.2 described the difficulties associated with this situation, showing how easily these mutual exclusion locks can cause deadlock if care is not taken. Even if this is appropriately handled, the packet arrival interrupt could (potentially) push packets to the buffer at the same (or a greater) rate than the timer handler is popping them,

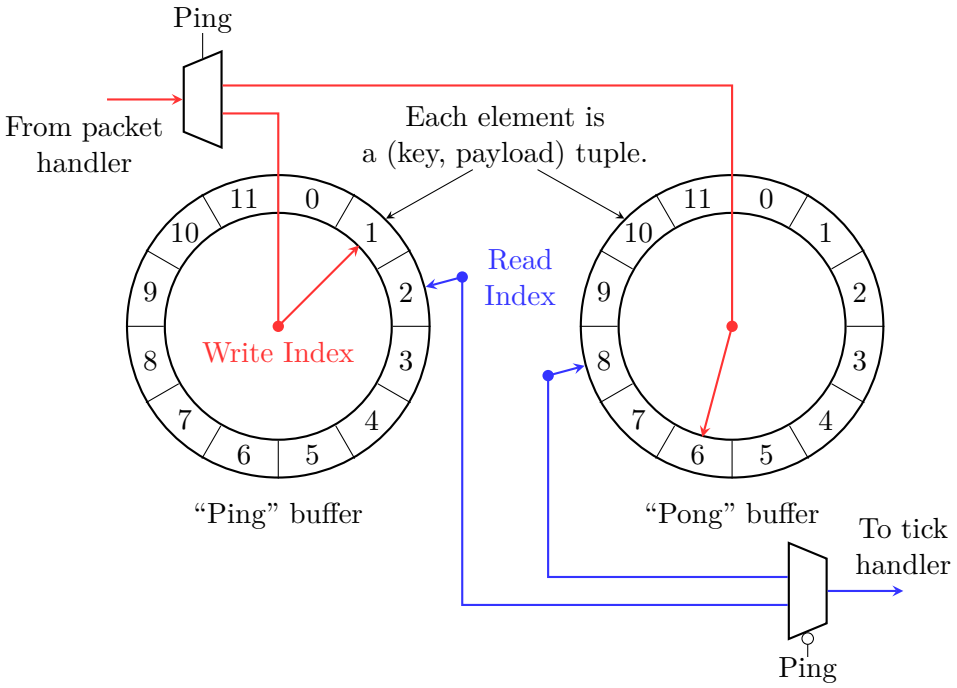


Figure 6.11: Visual representation of a 12-element "ping-pong" buffer.

which both inhibits all device updates on that core and prevents the timer tick handler from ever returning. The workload issued to the timer tick handler must therefore be appropriately bounded and free from contention.

Figure 6.11 shows the software architecture used to address these issues. There are two identically-sized circular buffers, named "ping" and "pong," which are exclusively written to or read from at any one time; i.e., if the "ping" buffer is currently being populated by the packet arrival handler, then the "pong" buffer is currently being emptied by the timer tick handler. Essentially, each buffer is an exclusive resource in the context of whichever handler is currently using it.

Each element of the buffer is a tuple containing the key (i.e., source address) and payload of the MC packet that triggered the packet arrival handler. The sole operation of the handler is simply to push this tuple into the active *write* buffer, which is fast enough to reduce the back-pressure applied to the network sufficiently far for the application to behave correctly.

The timer tick handler first swaps the active read and write buffers, disabling interrupts before the swap and re-enabling them afterwards. This the only part of the implementation that has the potential for contention. However, if a packet arrives before the timer tick handler has disabled interrupts, it is pre-empted and current write buffer grows, and if a packet arrives after the interrupts have been disabled, it is queued until the interrupts are enabled again; in both cases, the behaviour is correct. As interrupts are disabled for such a short time, any back-pressure that builds up has an insignificant effect overall.

Once the swap has completed and interrupts are enabled again, the timer tick handler enumerates the newly activated read buffer (thus bounding the workload), and invokes the IntHand function call that would usually be registered against the packet arrival interrupt for each key/payload pair. These *virtual packet arrivals* are all performed before any of the device updates are issued, and are thus indistinguishable from conventional packet arrivals from the perspective of the device handlers. Clearly this requires a slightly larger timer tick period than would otherwise be required, but it brings the important benefit of significantly improved handling of high volumes of packet traffic without perturbing the application model.

6.3.4 Downloading Data

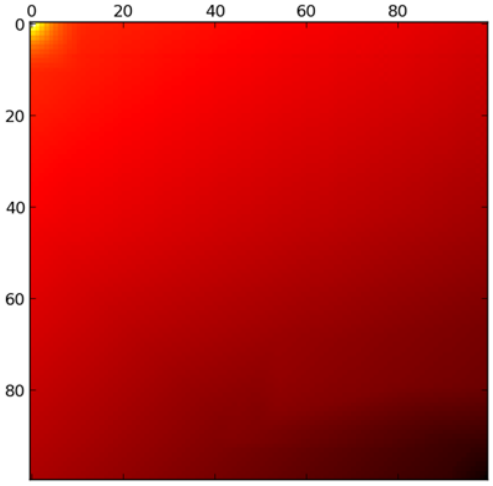
Downloading results from SpiNNaker applications is complicated by them being split between many cores. Core-local address spaces may, obviously, only be accessed by that particular core. Querying this space for current problem device values forces the worker cores to handle SCP commands while the application is running, which will degrade simulation performance in the best case. In the worst case, the commands will simply be ignored as the SCP interrupt priority is lower than the timer tick.

Instead, each problem device writes its state into an area of the node-local SDRAM designated by both core and problem device index. A "domain-specific post-processing" tool (Figure 6.6) tool waits for the application to terminate, after which it reads these blocks of data and assembles them into a contiguous file for analysis on the host machine. Reading arbitrary memory locations is included in the standard set of SCP commands.

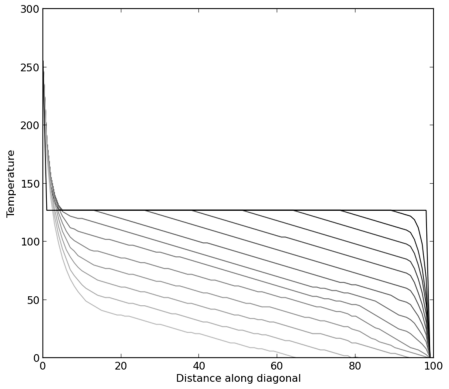
6.3.5 Results

Figure 6.12 shows the steady-state solution computed by the SpiNNaker implementation of Figure 6.2 and Algorithm 6.1 without any faults introduced. Figure 6.12b shows the convergence trajectory of the diagonal of Figure 6.12a. Darker lines show results from a lower value of k. It is clear from this plot that even when subject to uniform initial conditions (the darkest line) that the two constant value devices at $T_{0,0}$ and $T_{n,m}$ are sufficient to start the simulation.

Figure 6.13 shows the steady-state solution when 5% of the *problem devices* have been inhibited, which can be seen as the scattering of contrasting red cells Figure 6.13a. Despite the significant number of problem devices that refuse to update, the overall solution is still qualitatively correct. The numerical trajectory shown in Figure 6.13b is expectedly noisy, but still begins to settle down as k increases. Large spikes can be seen as the curve approaches the steady state. These are artefacts introduced by the disabled problem devices as their values have (deliberately) not changed over the course of the

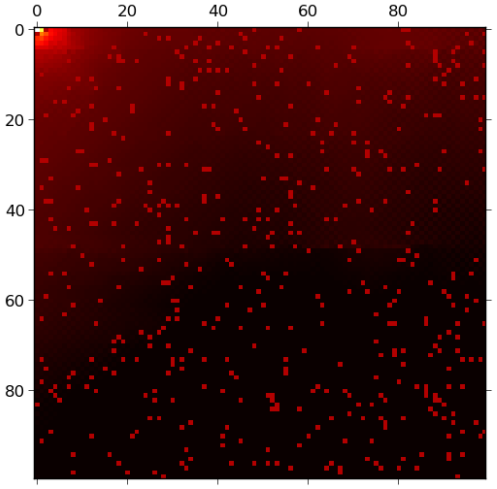


(a) Steady-state solution.

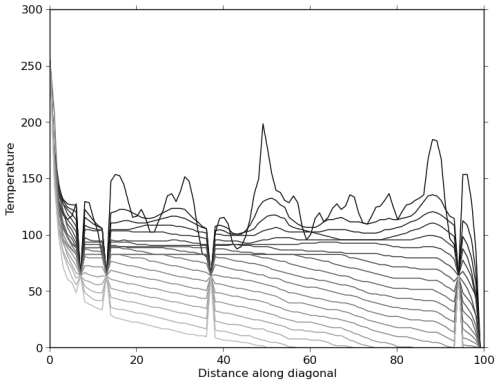


(b) Gradient along the the diagonal for increasing values of k; the lighter the grey, the larger the value.

Figure 6.12: SpiNNaker implementation results without any faults introduced.



(a) Steady-state solution.

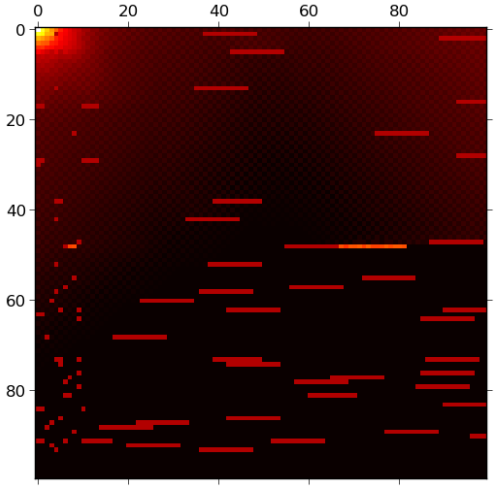


(b) Gradient along the the diagonal for increasing values of k; the lighter the grey, the larger the value.

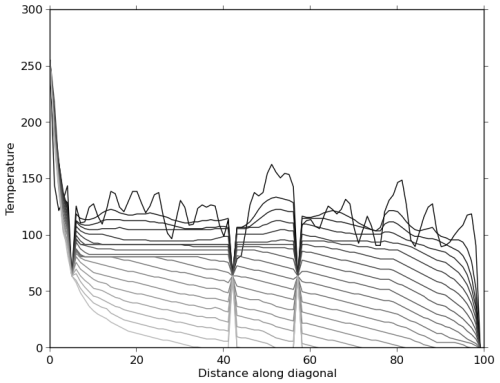
Figure 6.13: SpiNNaker implementation results with 5% of problem devices disabled.

simulation. However, the shape of the curve converges to the expected value regardless of these spikes.

Figure 6.14 shows the steady-state solution where 5% of the *cores* have been artificially disabled. Each core has around 15 problem devices mapped onto it, which creates the bar pattern as a consequence of the mapping algorithm in Loader. Despite the fault conditions being significantly more severe than in Figure 6.13, the qualitative validity of the overall solution is still clear—the effect of disabling 5% of the cores is simply to cut holes in the discrete model of the underlying physical system.



(a) Steady-state solution.



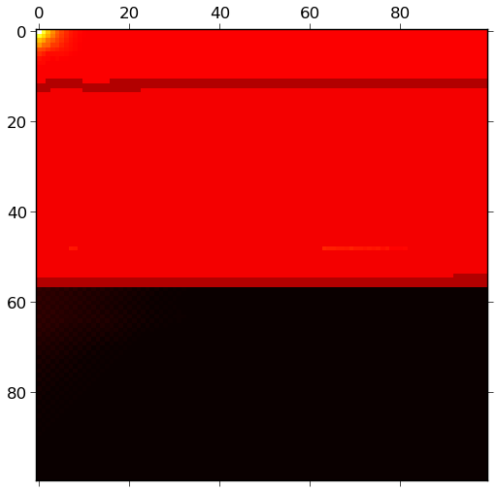
(b) Gradient along the the diagonal for increasing values of k; the lighter the grey, the larger the value.

Figure 6.14: SpiNNaker implementation results with 5% of cores disabled.

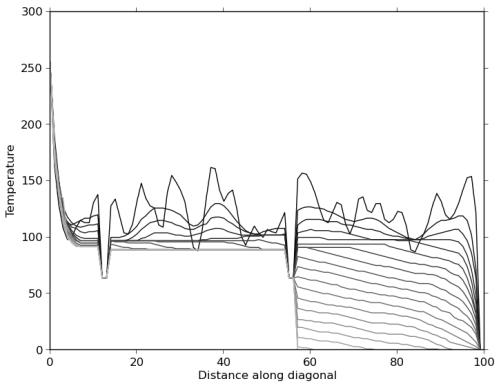
Figure 6.15 shows the steady-state solution with 5% of the *nodes* disabled (i.e., 2 on a SpiNN-103 machine). This has created a fault condition from which the application cannot recover. Disabling a node creates such a large group of disabled problem devices that the problem grid is partitioned twice. Within each partition, the solution is still correct according to the update rules, but it is clearly not qualitatively correct overall. The trajectory plot in Figure 6.15b shows that a steady-state solution is found (individually) in the three partitions despite the faults.

Constant value devices only exist in two of the three partitions: the sink device in the lowermost partition has drained all of the heat away as expected, and the source device in the uppermost partition has introduced sufficient heat to cause a gradient to appear. Without a sink device present, the entire partition should homogenise to the $T_{0,0}$ value (255°), which has clearly not happened here; there reasons for this are not clear, but it could be that the node failure has not completely separated the upper and central partitions, allowing some heat to flow around the edge. The central partition has homogenised because there are neither sources nor sinks in this area. This is to be expected because there is no notion of thermal reactance in the application, hence heat is never passively lost. Instead, the devices have computed the average of their initial conditions, which is a constant value.

Figure 6.16 shows how the wall-clock time changes as problem size increases on a SpiNN-103 machine with varying timer tick periods. For each period, scaling is largely independent of problem size which is expected, but still impressive. In comparison to a serial desktop machine, the compute time necessarily increases with problem size. This graph is a realistic illustration of the Amdahl vs. Gustafson argument posed in section 1.2; the serial computer fails to scale where SpiNNaker succeeds.



(a) Steady-state solution.



(b) Gradient along the the diagonal for increasing values of k; the lighter the grey, the larger the value.

Figure 6.15: SpiNNaker implementation results with 5% of nodes disabled.

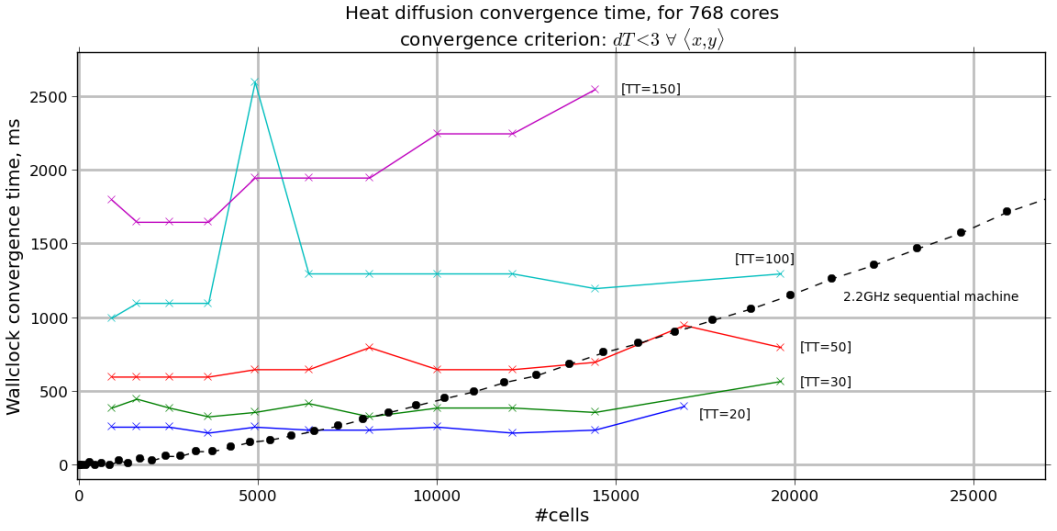


Figure 6.16: Comparison of application run-times with various timer tick periods on a SpiNN-103 against a conventional serial desktop machine.

Choosing an appropriate value for the timer tick is a non-trivial task because it depends on the expected computational requirement of the application. Recall that the purpose of the timer tick in this case is to help spread out the number of packets being emitted at any one time to prevent saturation of the network. Applications with lower communication loads may be able to function with smaller timer ticks, whereas applications with greater emphasis on communication will need to employ additional methods to help spread the load; tuning the timer tick is an empirical procedure.

6.4 Concluding Remarks

This chapter has shown how a non-neural application can function correctly within the neuromorphic framework provided of SpiNNaker. Converting the application into a form appropriate for SpiNNaker requires care, but it can also be viewed as taking the SPMD approach to the extreme, where each problem device forms an "atom" of the application [1]. Many numerical problems can be mapped onto this principle, so while SpiNNaker is not a general-purpose supercomputer, it can be used for applications outside of the neural domain.

However, the key point here is not that SpiNNaker, specifically, can be used for non-neural applications. It is that a computer with enough processors to make them a trivially available resource is indeed useful for solving real-world problems. Extreme-scale parallelism is inevitable, and applications structured in the way described in this chapter provide a way to effectively utilise them.

Along the way, reliability of computation has been considered at various levels. The SpinDiscover tool provides means to map applications around faults that are discoverable before the application even starts, and a correct formulation of the problem allows faults that develop at run-time to be tolerated in a similar manner to the neural applications for which SpiNNaker was designed [5, 7].

Clearly embracing massive parallelism is not a trivial task, but it is definitely tractable, and is indeed essential as machines continue to increase in size.

Chapter 7

Reliability

Modern computing systems place heavy emphasis on being reliable, and coping with failure gracefully becomes increasingly import as systems grow in size. IBM show a failure rate of around 0.08 faults/month/teraFLOP on their Blue Gene/L system [142]. Whilst this sounds perfectly reasonable, scaling this up to the exaFLOP regime leads to almost 2 faults/minute, which is not.

Some types of fault carry more severe consequences than others, and it is impossible to account for every conceivable failure mode in advance. Fault correction systems greatly aid the resilience in memory systems, but meaningful recovery from a serious fault is usually impossible for a computer architecture during software execution, even for high-reliability architectures such as the LEON-FT [143]. Nonetheless, SpiNNaker hardware has fault-tolerance built in at various levels, and the software designed to be run on the machine can be configured to make use of this capability.

7.1 Hardware

The complete SpiNN-106 machine contains around 57k nodes (i.e., 57k routers), 350k inter-node communication links, over 1M ARM968 processor cores, and approximately 7TB of distributed memory [35]. This section briefly outlines the broad categories of failure and then describes specific measures built into the system that ameliorate the fault consequences.

7.1.1 Cores

The first processor past the POST becomes the *monitor* core which is responsible for all node-level activities (described in section 3.2.1). This initial asynchronous race condition

naturally excludes cores that exhibit faults because they will not acquire an identifier—recall from section 7.2 that cores are labelled monotonically increasing from 0 (the monitor).

Any core can be disabled by the monitor core writing to the appropriate register in the system controller (shown on Figure 3.1). The *watchdog timer* (also shown on Figure 3.1) protects against the monitor core stalling by requiring periodic writes to indicate liveness. This is implemented as a decrementing counter that raises an interrupt on first counter expiry (allowing the monitor to execute recovery behaviours), and asserts the reset output on the second expiry (hard-restarting the entire node).

Detecting stalled worker cores requires a software protocol initiated by the monitor core. Each worker has a self-test interrupt that can be used to report liveness back to the monitor. Failed liveness reports cause the monitor to reset the specific core. Recovery from this type of fault is extremely difficult because both the software and state are lost; the most viable approach is to write applications that can tolerate core loss.

7.1.2 Interrupts

It is theoretically impossible to detect the *erroneous* invocation of an interrupt [7]. It is also infeasible to detect that an interrupt has failed to complete its task correctly, even if it has been invoked appropriately. As SpiNNaker is an interrupt-centric design, these faults can have severe consequences. Regular checkpoints can be used to detect that the application is not converging as expected, but identifying the core/node causing the issue is not sufficient to recover from it.

7.1.3 Routing Subsystem

Worker cores that have stalled will consequentially burden the communications network NoC because the router will continue to send packets to them. As the monitor core is capable of detecting and disabling faulty cores, it can also adjust the MC routing tables to prevent routing to them from even being attempted.

Nodes are particularly sensitive to failures in the router and/or the monitor core, and it is difficult to see how any meaningful recovery could be achieved here. Faults of this nature require a complete rediscovery of the machine (Chapter 5) and a re-mapping of the application (Chapter 4), which is not really a viable recovery mechanism.

7.1.4 Packet Parity

The lowest level of defence against faults protects against corrupted data. All packets feature a parity bit in the control byte (Figure 3.7) which is set by the core-local communications controller such that the packet has *odd parity*. If this bit is wrong at any stage of transmission, the router will drop the packet to prevent further propagation.

This drop is recorded in the router statistics registers [115, pgs. 50–51], and software interrogation of them ultimately cannot tell the difference between a packet being discarded in this manner and a packet being discarded due to back-pressure. The goal of the parity bit is simply to prevent any bad data from being propagated to the application software.

One bit of parity does not offer perfect protection, but the network itself is usually reliable enough for a single bit to be sufficient. Inspection of the SCAMP source reveals that system packets (existing NN and P2P traffic, not any added in the thesis) include a 4 bit 1's complement checksum in the payload of the transmitted packets to further protect against erroneous bit-flips.

7.1.5 Network Time Phase

Each router in a SpiNNaker system maintains the current *time phase* [115, pg. 42] which is a 2 bit field that regularly updates. Every multicast packet introduced by the core-local communications controllers has the T field of the header (see Figure 3.7) set to the current phase of the router in the node. Packets that are two or more phases old are discarded. The use-case for this behaviour is to protect against bad MC routing tables by preventing MC packets from flowing around the SpiNNaker torus indefinitely.

Figure 7.1 shows the time phase concept visually. T_n refers to some arbitrary time phase in which the packet is emitted, and T_{n+1} refers to the subsequent phase. The green route represents the first orbit of the torus and the red route represents the second. Packets are subject to the router time phase at the following points identified on the figure:

1. Packet is emitted with $T = T_n$ and is routed east.
2. Some time later, the system time phase, T_s, asynchronously advances to T_{n+1}.
3. The packet re-enters the source node and is routed according to the same routing entry as before, thereby creating a cycle.
4. As before the system time phase advances to $T_s = T_{n+2}$.
5. The packet enters the source node again, but is now discarded because $T_s - T \geq 2$.

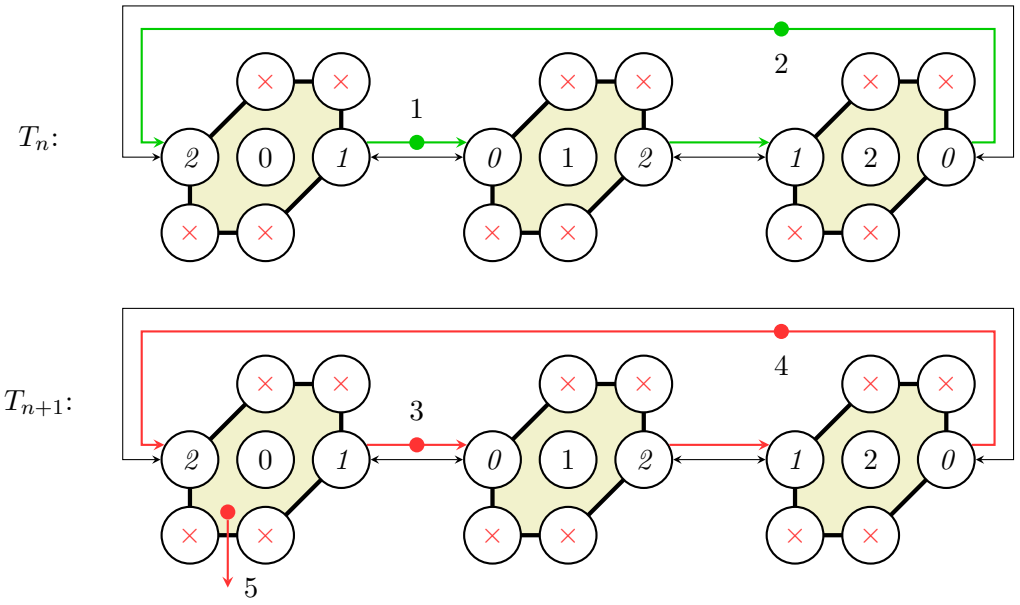


Figure 7.1: Packet phase prevents packets from flowing around the network indefinitely.

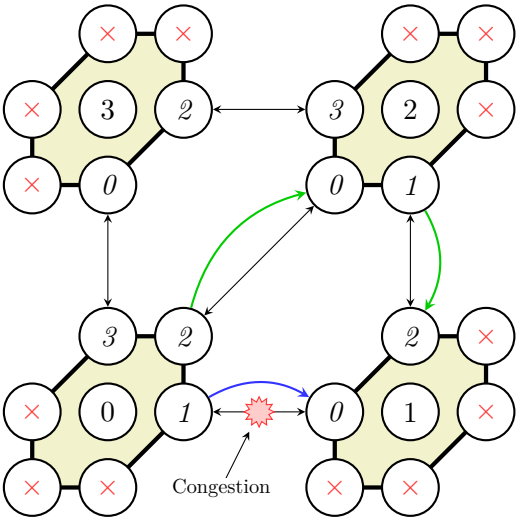


Figure 7.2: Emergency routing around a heavily congested link.

Ideally, the time phase update rate should be tuned according to the major radius of the torus. Without this tuning, smaller machines permit larger numbers of cycles.

7.1.6 Emergency Routing

Emergency routing was first mentioned in Chapter 3 as a method of dealing with particularly congested links. If the router cannot transmit an MC packet out of the required link because the back-pressure is too severe, it will attempt to *emergency route* via a neighbour. If the back-pressure on that link is also severe, the packet will be dropped.

Figure 7.2 illustrates this point. Node 0 wishes to send node 1 an MC packet using the blue route, but the back-pressure on the western port of node 1 prevents the transmission from completing. Instead, node 0 routes the packet via green route through node 2.

MC packets contain an emergency routing field in the header. Node 0 will set this field to indicate that the packet being sent out of the north-eastern port is being emergency routed and is therefore not actually destined for node 2. The router of node 2 reads the emergency routing field and redirects the packet to the original location, but alters the emergency routing field to reflect that the packet *has been* emergency routed. This allows node 1 to treat the packet as if it were received from the eastern port instead of the northern port.

Emergency routing is always performed in the *clockwise* direction. Thus restoring a packet to its original destination is simply a case of rotating anti-clockwise. A node receiving a packet that has been emergency routed (node 2 in this case) therefore does not need to inspect the contents of the packet beyond the emergency routing field.

This approach to fault-tolerance is predicated on the geometry of the system (SpiN-Naker) and cannot be applied to an arbitrary topology. Whilst emergency routing affords some tolerance of congested links, it does not extend to *faulty* links unless the traffic density is sufficiently low.

7.1.7 Dropped Packet Re-injection

When any router decides to drop a packet (for whatever reason), it is stored in a set of registers that completely describe the situation, and the 'router dump' interrupt is raised. These registers describe not only the full content of the packet, but also store the direction in which routing was attempted.

Any processor that listens for the interrupt can read these registers and push the data into a software queue for later re-injection. Typically, a worker core is dedicated to this task in applications that may have bursts of high packet flux because the interrupt *must be handled quickly* as a second dropped packet will overwrite the register contents. The timer tick of this worker core is used to clear the queue by re-injecting all of the packets that were dropped.

7.2 Software

7.2.1 Problem Device Placement

With accurate machine topology information obtained by the algorithms designed in Chapter 5, applications can be placed by Loader in a manner that reduces the number of problem edges that are mapped onto physical edges, thus reducing the volume of inter-node packets. Applications based on a uniform grid (such as the heat diffusion application in Chapter 6) are still conducive to this approach. The problem grid can be broken into multiple sub-grids of arbitrary size, which reduces the inter-node communication requirement to the boundaries of these sub-grids.

In the general case, packet flux can be kept minimal by simply grouping highly-connected portions of the problem graph into as few nodes as possible. Long-range routes are subject to more routers and hence have a greater chance of encountering emergency routing or being dropped and re-injected at a later time. Run-time efficiency (and, hence, reliability) are improved by better decisions made in the mapping stage.

7.2.2 Inherently Robust Applications

The most obvious approach to reliability is ensuring that the applications themselves tolerate a range of fault conditions, as with Chapter 6. This plays to the original design specification of SpiNNaker where applications are assumed to be resilient against packets not arriving. Numerical applications can be made tolerant to these sorts of faults through careful formulation.

An important aspect of the fault resilience of SpiNNaker is the nature and consequence of failures. In a generic electronic system, faults manifest in different ways: stuck-at, bridging, crosstalk, or some combination. In a conventional supercomputer, the effects of a core or communication hard fault are usually catastrophic. Checkpoint, rollback and recovery protocols [140] are a recovery mechanism of last resort, but are expensive and make assumptions about the nature of a failure. At almost every level (apart from the cores themselves) SpiNNaker is asynchronous, and a consequence of this is that a core or node can only realistically fail silently. This, in turn, means that the affected problem devices (those mapped to the failed core/node) effectively simply disappear, rather than continually inject incorrect data into the system.

This does not mean that the failed problem device ceases to have an effect on the numeric solution. Prior to the failure, copies of the problem device state are stored in the local state-space of its immediate neighbours (see Algorithm 6.1). Post site failure, these values will never change (EDGEHANDLER is never again executed with that particular `edge.type`—Algorithm 6.1).

In the context of a *relaxation* based simulation (of which the heat diffusion application in Chapter 6 is a canonical example), a fault represents the replacement of a passive component (model) in the system under simulation with an energy source. This takes the form of a potential source of whatever value the problem device state held at the point of failure (and may be positive or negative). This state perturbation—of a single problem device—may or may not have a negligible effect upon the numeric configuration of the solution (depending upon its position). This notwithstanding, the fault does not compromise the stability or *speed* of the overall solution process [7].

On a conventional architecture, physical failure compromises (usually fatally) the solution *algorithm*. On SpiNNaker, physical failure locally perturbs the *model*, not the solution technique. Refer back to section 6.3.5 where the solution remained qualitatively correct overall despite introducing quite severe faults (Figures 6.13 and 6.14). Sufficiently severe faults can perturb the model to the point where it no longer describes the original problem specification, but problem device computations remain locally correct even if the overall result has become meaningless (Figure 6.15). It is difficult to see how applications could be engineered to tolerate hardware failures of this severity.

7.2.2.1 Recovery

SpiNNaker can detect core/communication failures on the fly by use of the the 'maintenance ping' introduced in section 7.2.3. On detection of a fault, the simulation may be modulated by informing the problem devices logically adjacent to the silent area that their local state copies (described above) should be disregarded. This strategy is superior to a conventional (and expensive) checkpoint/rollback recovery regime [140], because the SpiNNaker solution trajectory is intrinsically self-healing—processing can continue from the point at which the fault was *detected* (not necessarily occurred) and will nevertheless re-converge to the *correct* solution.

Neural networks are inherently tolerant of data loss, which implies there exists a family of more general applications that rely on statistical packet arrival characteristics instead of absolute arrival. The difference between this and numerical applications is subtle. With numerical applications, missing packets may cause a delay in convergence, and any delay in convergence naturally eliminates the problem caused by the missing packet. An application that makes use of statistical packet properties would send potentially redundant data between problem devices to amortise the effect of any dropped packets. This is closer to neural applications where a spike in isolation is usually meaningless (or spurious), but a *chain* of spikes convey a signal.

7.2.3 Real-time Fault Detection

Detecting faults at run-time can be achieved by re-running any of the algorithms designed in Chapter 5 multiple times. Subsequent surveys take the form of a distributed 'ping' orchestrated by a traversal of the control tree in a similar manner to the "continuity algorithm" (section 6.3.2). Any changes detected are be reported to an external host machine tasked with monitoring the system.

Reacting to any faults detected in this manner is a non-trivial problem and requires re-distribution of problem data in the simplest case, and distributed recompilation of both problem device data and handlers in the most complex. Applications reacting dynamically to run-time faults is a "holy grail" of reliable distributed computing.

Chapter 8

Concluding Observations

Chapter 1 identified the need for new software techniques that can take advantage of the increasing number of readily-available processors in each new generation of computer hardware. Parallel programming is already a notoriously difficult practice, and large processor counts do not make current approaches any simpler.

The approaches to fault-tolerance and resilient parallel programming built up in Chapters 4–7 show that massive parallelism can be used to solve real-world problems despite the non-determinism of the underlying communications fabric. Interleaving appropriate fault-detection algorithms allows faults that have developed during execution to be sensibly handled.

Whilst these techniques originated to broaden the portfolio of problems that can be solved by SpiNNaker, they lay the ground-work for a new perspective on parallel software and parallel computing in general. This chapter presents some avenues of future work that can explore these new perspectives.

8.1 Contributing Back to SpiNNaker

8.1.1 Non-neural Infrastructure

Chapter 5 builds on the existing infrastructure to support more powerful parallel programming techniques, using the concept of the "control tree" to implement a barrier capability. The other algorithms presented in that chapter, along with the configuration aspects of the tool-chain presented in Chapter 6, use the control tree for barrier-like activities without using it as a barrier.

Extending the SpiNNaker API provided by SARK to understand the notion of a barrier further broadens the scope of problems that can be mapped on the infrastructure built

in this thesis. Consequently, more conventional parallel programming techniques such a scatter/gather [78] and potentially even map/reduce [144] could be constructed around the barrier.

Scatter/gather is perhaps the simplest extension of the barrier. *Scatter* distributes a particular value (or set of values) to all cores in the system. This can be trivially achieved by sequentially visiting each node of the control tree. Doing so in this manner limits the traffic to only a single direction across the machine fabric and out of a limited number of ports on each node. Contention is therefore completely avoided because cyclic traffic flows are not possible (the structure is a tree).

Gather is the complementary operation where data is collected back to the root. It is common for data collected in this manner to be subject to some function. For example, consider a set of problem devices, $\mathbb{P}$, which each produce a result, $R_p \forall p \in \mathbb{P}$; the gather operation can apply the `sum` function as results are gathered, leading to the root receiving

$$R = \sum_{p \in \mathbb{P}} R_p. \tag{8.1}$$

However, this technique is not necessarily limited to problem devices on the root node. *Any* problem device could request that a gather-`sum` is performed via an API call. This request bubbles up the control tree to the root, wherein the root issues a gather-`sum` request to all nodes via the control tree, and eventually R arrives back to the root, where it is forwarded back down the control tree to the node from which the request originated. P2P messages are used to remove the initial and final control tree traversals.

Map/reduce is much the same as scatter/gather, but map does not require the value being broadcast to be homogeneous. A smalll set of simple functions for use with Reduce/gather are implemented in SCAMP or SARK, but the concept can be extended to allow applications to register more complex domain-specific functions as appropriate.

A final extension enabled by the control tree is the ability to *pause applications at run-time*. This is more complicated than it sounds; recall that there is no global synchronisation in SpiNNaker, so the idea of a computational "now" is *only* defined by the barrier. Consider the case where an application is steadily applying a set of constants to a flow of data from some source (potentially external to the physical hardware). After the application starts, a new set of values for these constants becomes available. Instead of rebuilding the data-structures and re-uploading the application, all cores are issued with a *pause* message that flows down the control tree. Then a scatter/map operation (or a directed-load) distributes the new parameter values. Finally, an *unpause* message restores the application to normal working order.

8.1.2 Simplifying Application Development

Augmenting SCAMP and SARK with the features described above allows SpiNNaker to be the target of *productivity languages* that aim to reduce the complexity of writing parallel software. X10[1] is of particular interest here because it offers a tool-chain that generates C++ code designed to link against libraries that handle communications and message-passing. Typically it assumes the presence of either MPI or the Deep Computing Messaging Framework (DCMF) which is a variant of MPI designed for use on IBM Blue Gene supercomputers. SpiNNaker bindings based on these concepts can easily add to this repertoire, making SpiNNaker a viable target.

X10 is a high-level object-oriented language with asynchronous behaviour built into the language itself. In contrast to some of the language features mentioned in section 2.1.4 (e.g., the `async` and `await` keywords in C#), X10 has been designed with a multi-process model in mind, supporting both the SPMD and MPMD models. With careful implementation of the appropriate binding libraries, a language such as X10 can simplify writing SpiNNaker applications.

8.2 Fault Tolerance Techniques

8.2.1 Real-time Fault Discovery

Chapter 7 introduced the idea of running the discovery algorithms presented in Chapter 5 periodically during run-time to discover new faults that arise. The simplest course of action to take is merely to report the presence of a new fault to the host computer so that the application can be rebuilt, re-uploaded, and restarted. The **omega** was introduced in section 5.6.2 illustrating that multiple disjoint state-machines can coexist in an event-driven architecture without issue, supporting the notion of the algorithms running alongside an application.

However, section 8.1.1 described using the control tree to pause the application at run-time. Outside of SpiNNaker, this concept can be achieved in software using similar techniques. Once paused, a snapshot of the state of the cores around the fault can be downloaded and used to recompile only the part of the application affected by the fault rather than the entire thing. The state snapshot can be restored to the nodes (subject to any data migration required by problem devices that have been moved), and then the application can be unpaused.

This approach is not necessarily applicable to all types of parallel application, but those that are numerically similar to the heat diffusion application outlines in Chapter 6 will

[1] http://x10-lang.org/

recover from erroneous values caused by this repair. In the simplest case, the fault is a malfunctioning port, which can be applied to all applications because only the adjustment of problem edges is required to bypass the port.

8.2.2 Orthogonal Encoding Schemes

Following on from the above, tolerance of core and potentially even node faults might be achievable if the data for a node can somehow be reconstructed from data stored in neighbouring nodes. This is somewhat similar to RAID arrays which provide a degree of robustness against data-loss by encoding data across multiple hard-drives.

If data for a particular node was split into smaller (potentially compressed) segments distributed between neighbours, then a node failing during the execution of an application can potentially be dealt with inside the machine rather that requiring an external recompilation. As with the offline method described above, the application would need to be globally paused using the barrier method, but then nodes around the fault could dynamically increase the number of problem devices present to absorb those lost on the failed node. Obviously the exact state of the devices at the point of failure cannot be reconstructed unless additional data is shared, but for numerical applications that converge with a stable trajectory this is not an issue.

Tolerating core faults is easier provided that problem devices are under-allocated to ensure there are always spare cores. Copies of the core-local (or process-local, in a general sense) data must reside in shared memory so that the new core can take over the workload.

8.3 Parallel Software Synthesis

8.3.1 Functional Decomposition

To produce programs, compilers first build a graph of the operations described in the source code. This graph passes through several optimisation stages before being synthesised into the appropriate transfer language for that layer. For example, `g++` compiles C++ code into a Lisp-inspired intermediary language called Register Transfer Language (RTL), which is then synthesised into assembly instructions for the target platform.

Problem graphs (see Chapters 4 and 6) can be similar to the graphs that lead to RTL, but offer a far higher-level description of the software behaviour. This cross-over provides a method of synthesising event handlers—both for SpiNNaker and in a general “atomic computing” sense.

For best results, a model of the data flow through the program is also required so that workloads can be transformed into parallel segments where possible. Functional programming languages are a natural route for this because, typically, functional programs are compositions of general concepts that are applied to data. If a workload has a set of functions that are applied in a loop with a constant termination criterion, it can be trivially unrolled and synthesised into as many problem devices as required to obtain the desired performance.

A more sophisticated approach is to examine the data-dependency graph of a program, which contains the relationship between all the variables in all scopes of the program. Functions that perform several discrete operations on disjoint variables can be split into several problem devices operating in parallel.

This level of decomposition leads to very fine-grain parallelism that forms the basis of the "atomic computing" principle. Tools like ROCCC[2] convert 'C' programs into highly parallel implementations that can be synthesised into FPGAs to yield significant performance increases; thus providing a good foundation for this avenue of enquiry.

8.3.2 Massively Reconfigurable Computing

SpiNNaker shares many similarities with programmable hardware. Hardware Description Languages (HDLs) are inherently parallel and provide a suitable front-end for describing "atomic computing" applications. HDL designs are described as a set of modules with signal paths that link them all together. A description of this form blends the handler descriptions and problem graphs into a single entity.

Perhaps the most powerful aspect deriving from this is local reconfiguration at runtime. Applications structured in this way naturally form a graph of interconnected components that may be swapped for other implementations as required, triggered by a signal from within the application itself. Using the application-pause concept described in section 8.1.1, the application can be halted whilst individual programs are swapped for more appropriate ones based on new information, then the barrier can be released to resume execution.

This concept is perhaps most useful for dataflow programs which stream large volumes of data through the problem graph. Coupled with some basic machine learning techniques, this leads to highly self-adaptive and potentially self-repairing distributed applications.

[2] http://roccc.cs.ucr.edu/

8.4 Closing Comments

This thesis demonstrates the following:

- Contributions to the SpiNNaker tool-chain for non-neural applications (Chapter 4):
  - The *Uploader* tool [127] writes application data and routing tables mapped by Loader into a target SpiNNaker machine, along with appropriate compiled binaries. The communications subsystem that underpins this provides an API that can be used separately to Uploader, enabling programs running on conventional hardware to communicate with SpiNNaker and applications running on it.
  - The *IntHand* API [127, pgs. 17–23] is compiled into SpiNNaker applications to augment the packet arrival and timer tick interrupts with the data mapped by Loader, by traversing the mapped data-structures.
- Discovery algorithms that perform a complete fault-survey of an arbitrary SpiNNaker-like system (Chapter 5) [3, 4]. Specific contributions include:
  - The fault survey of the SpiNN-104 hardware shown in Figures 5.47 and 5.48, subjecting the discovery algorithms to a real-world SpiNNaker environment.
  - Discovering the SpiNNaker machine topology whilst also assigning unique node labels (used to support P2P and MC communication) and constructing the *control tree*, which is used to support barrier-like behaviour (section 5.5).
  - Overcoming a shortcoming present in the existing node labelling scheme (section 5.5), benefiting *all* SpiNNaker applications, not just those in the non-neural domain.
- A demonstration application (Chapter 6) showing how to structure non-neural software so that fine-grained parallelism can be achieved on SpiNNaker [1, 5, 7], which has led to the following:
  - The *SpinDiscover* tool combines the discovery algorithms (Chapter 5) with a P2P table configuration algorithm (section 6.3.1), and downloads the *complete* topological description of the target SpiNNaker machine (section 6.3.2) to allow Loader to map applications to specific hardware.
  - A "ping-pong" buffering technique (section 6.3.3) that reduces the back-pressure felt by applications with a high packet flux.
  - An assessment of the reliability mechanisms available in SpiNNaker that allow non-neural applications to function on the neuromorphic hardware, provided they adhere to a set of identified criteria (Chapter 7).

Appendix A

Published Papers

This work has contributed to the following publications:

1. A. Brown, R. Mills, J. Reeve, *et al.*, "Atomic computing – a different perspective on massively parallel problems," in *VOLUME 25: PARALLEL COMPUTING: ACCELERATING COMPUTATIONAL SCIENCE AND ENGINEERING (CSE)*, Munich, Germany, 2013, pp. 334–343. DOI: 10.3233/978-1-61499-381-0-334

2. K. Dugan, A. Brown, J. Reeve, *et al.*, "Interconnection system for the SpiNNaker biologically inspired multi-computer," *IET Computers & Digital Techniques*, vol. 7, no. 3, pp. 115–121, May 2013, ISSN: 1751-8601. DOI: 10.1049/iet-cdt.2012.0139

3. K. J. Dugan, J. S. Reeve, and A. D. Brown, "Self-discovery Algorithms for a Massively-Parallel Computer," in *ADAPTIVE 2013, The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications*, Valencia, Spain, May 2013, pp. 36–39, ISBN: 978-1-61208-274-5

4. K. J. Dugan, J. S. Reeve, A. D. Brown, *et al.*, "Comparing topological discovery methods for a massively parallel computer," in *ACACES 2014 - ADVANCED COMPUTER ARCHITECTURE AND COMPILATION FOR HIGH-PERFORMANCE AND EMBEDDED SYSTEMS*, Fiuggi, Italy: HiPEAC, 2014, pp. 213–216

5. A. D. Brown, R. M. Mills, K. J. Dugan, *et al.*, "Reliable computation with unreliable computers," in *DESIGNING WITH UNCERTAINTY WORKSHOP*, York, 2014

6. A. D. Brown, S. B. Furber, J. S. Reeve, *et al.*, "SpiNNaker - programming model," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1769–1782, 2014, ISSN: 0018-9340. DOI: 10.1109/TC.2014.2329686

7. A. D. Brown, R. Mills, K. J. Dugan, *et al.*, "Reliable computation with unreliable computers," English, *IET Computers & Digital Techniques*, vol. 9, no. 4, pp. 230–237, Jul. 2015, ISSN: 1751-8601. DOI: 10.1049/iet-cdt.2014.0110

Appendix B

Booting Software on SpiNNaker

B.1 Scatter Load Files

Scatter load files are an important part of the SpiNNaker application compilation process because they essentially describe the memory map for the code. A scatter load file comprises multiple images, each containing multiple sections with some base address. SpiNNaker applications contain only a single image, but usually define (at least) three sections within. Listing B.1 shows an example scatter load description (a full description of the file format is given in reference [145]).

```
1 IMAGE_NAME 0
2 {
3     SECTION1 <address> [<size>]
4     {
5         some_object.o (some_area, <flags>)
6         * (<flags>)
7     }
8 }
```

Listing B.1: Sample scatter load file placing `some_object.o` in `SECTION1`.

The names adopted in the example above are fairly arbitrary and can be any alphanumeric characters provided that the first is non-numeric. The number following `IMAGE_NAME` is the base address for the rest of the image, which is useful if the image is going to be loaded at a fixed location on the target system. For SpiNNaker, this value should remain 0 so that the images can be moved around memory while retaining relative offsets elsewhere within the image.

`SECTION1` is a piece of code or data that will eventually be loaded into memory. The base address is required but the size seems to be optional. Within this section, multiple object files can be specified to give some kind of order of the image. An object file can contain multiple *areas* which are the units actually placed by the linker. When specifying

areas of object files, the most typical flags are +FIRST and +LAST which allow them to be aligned to the section in which they are defined.

The * object is a catch-all for any unspecified code. All objects passed to the linker that do not have an entry in a section will be placed in the most appropriate catch-all entry. Flags for these entries may be any of the following: +RO, +RW, or +ZI. +RO refers to read-only data that typically corresponds to executable code, +RW marks the section as read-write and therefore suitable for data and pre-initialised variables, and +ZI means zero-initialise and will cause the section to be initialised to all zeroes. +RO and +RW are mutually exclusive, and +RW sections are usually +ZI as well.

```
1 ARM_LIB_HEAP 0x00400000 EMPTY -0x1000
2 {
3 }
```

Listing B.2: Specifying a memory region to hold the heap memory.

The sizes of sections can be negative as well as positive; for example, Listing B.2 creates a section occupying the 4kB region *below* 0x00400000 (i.e., 0x003FF000–0x00400000) that will be EMPTY. This flag shows that the section can be used for a growable heap by the ARM standard library. There are three such sections named ARM_LIB_HEAP, ARM_LIB_STACK, and ARM_LIB_STACKHEAP that can reserve sections for the standard library. In any case, the heap grows from the bottom of an area upwards, and a stack grows in the opposite direction.

B.1.1 Defining Sections in Code

In ARM assembly language files, an area is opened with the following directive:

```
area    ‹name›, ‹options›
```

For code areas, readonly and code are the most common options which shows that it should be put in a +RO region of the scatter file and that it contains 32-bit aligned data or code. code16 and code32 can be used to switch between Thumb and ARM instruction sets and an appropriate alignment. Alignment can be specified with the align=‹bytes› directive. Finally, the root section of assembly code should follow an entry directive which marks the file as an entry point. This is why compiling applications that depend on SARK produced the multiple entry-points warning; there are multiple entry areas.

Specifying that 'C' code should exist in various named areas can be achieved in two ways, the simplest being shown in Listing B.3. Functions decorated by a section attribute will be placed in the readonly region of the named section [146]. However, variables decorated may appear in any region depending on how the variable has been

```
// Marking section in the prototype.
void SomeFunction(...)
    __attribute__((section("section_name")));
void SomeFunction(...)
{
    // Some stuff
}

// Marking section in the definition
__attribute__((section("section_name")))
void SomeOtherFunction(void)
{
    // Some stuff
}

// Variables have a slightly different syntax:
int SomeVariable __attribute__((section("section_name")))
    = 90;
```

Listing B.3: Mapping 'C' functions and variables to specific scatter load regions.

defined/declared [147]. Normal read-write variables (as above) will appear in the `+RW` region, `const` variables will be in the `+RO` region, and variables defined as follows will be in the `+ZI` region:

```
int SomeArray[90] __attribute__((
    section("section_name"), zero_init));
```

Parts of C/C++ source files can be divided into sections using the pragma directive [148] as shown in Listing B.4. There are three section types that can be specified with this directive: `rodata`, `rwdata`, and `code`.

```
#pragma arm section rwdata = "foo"
int SomeVariable = 9;       // In foo's data part
const int SomeConst = 1;    // In default rodata region
int SomeFunc (...) {        // In default code region
    // Some Stuff
}
#pragma arm section rwdata  // Resets to default rwdata region
```

Listing B.4: Dividing source code into sections using the `#pragma` directive.

B.1.2 Linker symbols

The linker defines a set of symbols for each section defined in the scatter load file according to the following rule: `Image$$‹region›$$‹section›$$‹attr›`. `‹region›` may take any name that has been defined in the scatter load file itself, using the listing at the

start of this section as an example: `Image$$SECTION1$$...` references the `SECTION1` of the image. `<section>` may be one of `RO`, `RW`, or `ZI` as previously explained, and `<attr>` may be one of `Base`, `Length`, or `Limit` as defined in Table B.1.

Symbol Attribute	Description
Base	Base address of the region, relative to the base address of the image file, as defined in the scatter load file.
Length	Length of the region in bytes (note: this excludes the `ZI` section if this references the region itself.)
Limit	Address of the byte immediately following this region; i.e., `Base + Length`. As with `Length`, this value does not include any `ZI` part of the region.

Table B.1: Description of the linker symbol attribute parts.

A top level region/section may also be referenced, so `Image$$SECTION1$$Base` and `Image$$SECTION1$$RO$$Base` are both equally valid and may point to different parts of the image.

These symbols can be accessed in both ARM assembly (Listing B.6) [149] and C/C++ (Listing B.5) [150] using specific syntaxes. In assembly, the imported symbol *must* use the same bar notation used to import the symbol.

```
1 // By Value:
2 extern unsigned int Image$$SECTION1$$ZI$$Limit;
3 // By Reference:
4 extern void* Image$$SECTION1$$ZI$$Limit;
```

Listing B.5: Accessing the linker symbols from within 'C' and C++.

```
1 import |Image$$SECTION1$$ZI$Limit|
2 ; Or
3 import ||Image$$SECTION1$$ZI$Limit||
```

Listing B.6: Accessing the linker symbols from within ARM Assembly.

B.2 ROM Boot-loader

The system boot software resides in the chip-local boot ROM and is executed automatically after power-up. A scatter load file (Appendix B.1) is used to define the various regions of the code, which are loaded into the correct parts of memory using standard routines supplied by ARM. Some of the code (such as the initialisation and program copier) remain in the ROM and are executed from there, whereas the rest of the executable code is copied to the ITCM and pre-initialised variables are copied to the DTCM. The ARM run-time invokes the 'C' `main` function after the copying has taken place.

`main` first initialises and test the various processor-local peripherals before engaging in the monitor arbitration process. The winner is then responsible for initialising and testing the chip-local peripherals (RAMs, router, watchdog timers), while the losers spin on bit 4 of the 'misc control' register (`r14`) [115, pg. 74] the system controller. The newly allocated monitor processor sets this bit and then *all* processors initialise their local timers and interrupt controllers, set their 'OK' flag in system controller register r4 [115, pg. 70], and then enter the wait-for-interrupt state.

Some important points to note here are:

1. The image copier responsible for loading and executing uploaded programs *remains* in the boot ROM and runs from there, making it impossible to write an image over the top of the copier.
2. All application cores effectively halt in the WFI state and may be woken up by an interrupt from the system controller (most probably caused by the monitor processor) or by a multi-cast packet arrival in the router.
3. The monitor core starts a timer with a period of around 1ms. Each time this timer expires, the monitor resets the chip-local watchdog timer and updates a measure of the elapsed time. Two separate periods are used for flashing an LED and also emitting a test Ethernet frame.
4. Entering the WFI state is achieved using the system control coprocessor [151]. The `MCR` and `MRC` instructions move data into and out-of the coprocessor registers respectively. When the coprocessor enters is instructed to enter the WFI state, the main processor enters a low power state and the AHB write buffer is emptied. The processor will be woken up even if interrupts are disabled, however, if they are enabled then the interrupt handler is *guaranteed* to execute before the instruction following the `MCR`.

Leaving the WFI state can be caused by incoming Ethernet packets, system controller interrupts, and/or incoming NN or P2P packets. The most relevant to this document is obviously the Ethernet packet arrival which eventually hands over control to application code. Specific boot packets are used that include an instruction field. Instructions are as in Table B.2.

The *entry point offset* of the above table could be a somewhat misleading name as it should point to the first byte of a binary image that should be executed. Usually this will be the APLX table (Appendix B.3) for the image that is added to the start of the image using scatter load files (Appendix B.1), but in the case of SCAMP, this will be the APLX-self extractor (Appendix B.4). The boot-loader passes control to the address in operand 3, therefore it is **required** that this portion of the image be executable code and not data.

Instruction	Value	Operand 1	Operand 2	Operand 3
`FF_START`	1	*(Unused)*	*(Unused)*	Number of 256 word blocks being sent.
`FF_BLOCK_DATA`	3	Word count in bits [15:8]; block number in [7:0]	*(Unused)*	*(Unused)*
`FF_CONTROL`	5	1	*(Unused)*	Entry point offset

Table B.2: Boot commands and their operands as used by the Ethernet boot.

B.3 APLX Images

An APLX image (abbreviation unknown) is assembled by placing an *APLX table* at the start of a compiled binary image using scatter load files (Appendix B.1). The ARM linker, `armlink`, outputs an ELF image that can be converted into multiple formats using the `fromelf` command that ships with it. As the layout of the image has been guaranteed by the scatter-load file, the following command can be used to strip off the ELF header and leave the binary data:

```
fromelf <compiled.elf> --bin --output <compiled.bin>
```

Usually these files have the `.aplx` extension but this is only for convenient identification. The APLX header format is a table of arbitrarily many rows with four word-fields. As with many SpiNNaker-related formats, the first of these four words is an instruction and the remaining three are operands specific to each command—summarised in Table B.3, and explained in Table B.4.

Instruction	Value	Operand 1	Operand 2	Operand 3
`APLX_ACOPY`	1	Destination address	Source address	Length in bytes
`APLX_RCOPY`	2	Destination address	Source address relative to the *start* of this entry	Length in bytes
`APLX_FILL`	3	Destination address	Length in bytes	Fill value
`APLX_EXEC`	4	Program counter	*(Unused)*	*(Unused)*
`APLX_END`	`0xFFFFFFFF`	*(Unused)*	*(Unused)*	*(Unused)*

Table B.3: APLX commands and associated operands for the APLX table.

Instruction	Description
APLX_ACOPY	Copies a block of data from the source (absolute) to the destination address (absolute).
APLX_RCOPY	Copies a block of data from the source (relative to the *start* of the APLX record) to the destination address (absolute).
APLX_FILL	Initialises an area by filling it with a defined value.
APLX_EXEC	Loads the program counter with the specified value, passing off execution to that portion of the loaded image.
APLX_END	Optional sentinel value to mark the end of the table.

Table B.4: Description of the APLX commands.

If the instruction field of the table is not one of the above values, or is APLX_END then the APLX loader will halt and hence the processor will need to be reset (which may be automatic if the Watchdog timer is in effect).

```
1       area    ‹name›, readonly, code
2
3       import  ‹various linker constants›
4
5   ‹aplx_table_label›
6       ; Row 1 - copy some code
7       dcd APLX_RCOPY
8       dcd ‹some address›
9       dcd ‹some address calculation - relative to start of record›
10      dcd ‹length in bytes›
11
12      ; Row 2 - execute the code
13      dcd APLX_EXEC
14      dcd ‹some address›
15      dcd 0
16      dcd 0
17
18      end
```

Listing B.7: APLX table construction in ARM Assembly.

The table itself must be constructed using ARM assembly as shown by Listing B.7. The area name is fairly arbitrary but must be used in the scatter load file to assemble the image correctly. Throughout the table, labels can be used to simplify the relative address calculations. No sentinel value is required because control is passed to the given address immediately after the APLX_EXEC command is processed. An APLX image can then be constructed by the linker with a scatter load file similar to that shown by Listing B.8.

When the linker produces the output ELF image file, the APLX_TABLE_NAME section will be placed at the start because it appears as the first section. ITCM_SECTION_NAME will follow and will contain the majority of the executable code. Specifying both sections

```
MY_IMAGE_NAME 0
{
    APLX_TABLE_NAME 0 OVERLAY
    {
        my_aplx_table.o (<area name from assembly file>, +FIRST)
    }

    ITCM_SECTION_NAME 0 OVERLAY
    {
        object_containing_entry_point.o (<entry point area>, +FIRST)
        * (+RO) ; All unspecified code modules
        ; Any other constraints
    }

    DTCM_SECTION_NAME <address in SpiNNaker mem. map> <size>
    {
        * (+RW) ; Use for read/write data (i.e. pre-initialised variables)
        * (+ZI) ; Zero-initialise the remainder of the section
    }
}
```

Listing B.8: Scatter load file to appropriately construct an APLX image.

as `OVERLAY` means that they can share an address space and also prevents the standard ARM-routines from loading an image. Instead, the linker assumes that there is an *overlay manager* present somewhere in the image that will sort these sections out. This is the purpose of the APLX tables and loaders.

Both SCAMP and SARK include an APLX loader that is copied to the very top of the instruction memory in the hopes that an APLX copy will not overwrite the copier itself. The boot-loader avoids this issue because the copier remains in the SRAM for the duration of the copy process. The simplest method of constructing an APLX table is to use the linker symbols defined in Appendix B.1.

B.4 Self-extracting APLX images

SCAMP has been designed to be executed by the boot-loader which has no understanding of the APLX format, however it is still compiled as an APLX image so that it can (in principle) be loaded by SARK if required. A small self-extractor is prepended to the main executable APLX image.

This bootstrap code can be executed directly (and will be if it is at the start of the image) and begins by copying the APLX loader to the top of the instruction memory. Execution is then passed to the freshly copied APLX loader and the SCAMP APLX table is processed identically to any other APLX image file.

B.5 SpiNNaker System Software

Both SCAMP and SARK include APLX loaders that behave identically to the self-extracting preamble. The loader is copied to the upper 64 bytes of the instruction memory and then execution is handed over. Obviously these function calls never return because control is passed.

SCAMP is compiled with both the self-extractor and a complete APLX table in the correct place. However, SARK is a library and merely contains the required information. A scatter load file is essential for all application code because it correctly places SARK *around* the application code. A typical SARK scatter load file is shown in Listing B.9.

```
1  APLX_IMAGE_NAME 0
2  {
3      APLX 0 OVERLAY
4      {
5          spin1_api_lib.o (sark_aplx, +FIRST)
6      }
7
8      ITCM 0 OVERLAY
9      {
10         spin1_api_lib.o (sark_init, +FIRST)
11         * (+RO)
12         spin1_api_lib.o (sark_align, +LAST)
13     }
14
15     DTCM 0x00400000
16     {
17         * (+RW)
18         * (+ZI)
19     }
20
21     ARM_LIB_STACKHEAP +0 EMPTY 0x1000
22     {
23     }
24 }
```

Listing B.9: Scatter load file for applications linking against SARK.

Unlike other scatter load files, the names in this one are *fixed* as above. This is because the linker symbols are already compiled into the partially linked image `spin1_api_lib.o` and hence having different section names in the scatter load file will prevent the required linked symbols from existing.

Bibliography

[1] A. Brown, R. Mills, J. Reeve, K. Dugan, and S. Furber, "Atomic computing – a different perspective on massively parallel problems," in *VOLUME 25: PARALLEL COMPUTING: ACCELERATING COMPUTATIONAL SCIENCE AND ENGINEERING (CSE)*, Munich, Germany, 2013, pp. 334–343. DOI: 10.3233/978-1-61499-381-0-334.

[2] K. Dugan, A. Brown, J. Reeve, and S. Furber, "Interconnection system for the SpiNNaker biologically inspired multi-computer," *IET Computers & Digital Techniques*, vol. 7, no. 3, pp. 115–121, May 2013, ISSN: 1751-8601. DOI: 10.1049/iet-cdt.2012.0139.

[3] K. J. Dugan, J. S. Reeve, and A. D. Brown, "Self-discovery Algorithms for a Massively-Parallel Computer," in *ADAPTIVE 2013, The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications*, Valencia, Spain, May 2013, pp. 36–39, ISBN: 978-1-61208-274-5.

[4] K. J. Dugan, J. S. Reeve, A. D. Brown, and S. Furber, "Comparing topological discovery methods for a massively parallel computer," in *ACACES 2014 - ADVANCED COMPUTER ARCHITECTURE AND COMPILATION FOR HIGH-PERFORMANCE AND EMBEDDED SYSTEMS*, Fiuggi, Italy: HiPEAC, 2014, pp. 213–216.

[5] A. D. Brown, R. M. Mills, K. J. Dugan, J. S. Reeve, and S. B. Furber, "Reliable computation with unreliable computers," in *DESIGNING WITH UNCERTAINTY WORKSHOP*, York, 2014.

[6] A. D. Brown, S. B. Furber, J. S. Reeve, J. D. Garside, K. J. Dugan, L. A. Plana, and S. Temple, "SpiNNaker - programming model," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1769–1782, 2014, ISSN: 0018-9340. DOI: 10.1109/TC.2014.2329686.

[7] A. D. Brown, R. Mills, K. J. Dugan, J. S. Reeve, and S. B. Furber, "Reliable computation with unreliable computers," English, *IET Computers & Digital Techniques*, vol. 9, no. 4, pp. 230–237, Jul. 2015, ISSN: 1751-8601. DOI: 10.1049/iet-cdt.2014.0110.

[8] G. Moore, "Cramming More Components Onto Integrated Circuits," *Electronics*, vol. 38, no. 8, pp. 114–117, Jan. 1965, ISSN: 0018-9219. DOI: 10.1109/JPROC.1998.658762.

[9] G. E. Moore, "Progress in digital integrated electronics," in *INTERNATIONAL ELECTRON DEVICES MEETING*, vol. 21, 1975, pp. 11–13.

[10] M. Santarini, "Xilinx Ships Industry's First 20-nm All Programmable Devices," *Xcell Journal*, no. 86, pp. 8–15, 2014.

[11] J. Soat. (2014). Maximizing Performance from the Bottom Up, [Online]. Available: https://www.oracle.com/servers/sparc/sparc-innovation.html (visited on 02/13/2016).

[12] S. E. Thompson and S. Parthasarathy, "Moore's law: the future of Si microelectronics," *Materials Today*, vol. 9, no. 6, pp. 20–25, Jun. 2006, ISSN: 13697021. DOI: 10.1016/S1369-7021(06)71539-5.

[13] K. Rupp. (2015). 40 Years of Microprocessor Trend Data, [Online]. Available: https://www.karlrupp.net/2015/06/40-years-of-microprocessor-trend-data/ (visited on 02/14/2016).

[14] H. Sutter. (2005). The free lunch is over: A fundamental turn toward concurrency in software, [Online]. Available: http://www.gotw.ca/publications/concurrency-ddj.htm (visited on 02/14/2016).

[15] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-c. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, "TILE64™ Processor: A 64-Core SoC with Mesh Interconnect," in *2008 IEEE INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE - DIGEST OF TECHNICAL PAPERS*, San Francisco, CA: IEEE, Feb. 2008, pp. 88–89, ISBN: 978-1-4244-2010-0. DOI: 10.1109/ISSCC.2008.4523070.

[16] D. Fick, R. G. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, T. Mudge, D. Sylvester, and D. Blaauw, "Centip3De: A 3930DMIPS/W configurable near-threshold 3D stacked system with 64 ARM Cortex-M3 cores," in *2012 IEEE INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE*, IEEE, Feb. 2012, pp. 190–192, ISBN: 978-1-4673-0377-4. DOI: 10.1109/ISSCC.2012.6176970.

[17] D. Fick, R. G. Dreslinski, B. Giridhar, G. Kim, S. Seo, M. Fojtik, S. Satpathy, Y. Lee, D. Kim, N. Liu, M. Wieckowski, G. Chen, T. Mudge, D. Blaauw, and D. Sylvester, "Centip3De: A Cluster-Based NTC Architecture With 64 ARM Cortex-M3 Cores in 3D Stacked 130 nm CMOS," English, *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 104–117, Jan. 2013, ISSN: 0018-9200. DOI: 10.1109/JSSC.2012.2222814.

[18] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, *et al.*, "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *SOLID-STATE CIRCUITS CONFERENCE DIGEST OF TECHNICAL PAPERS (ISSCC), 2010 IEEE INTERNATIONAL*, vol. 9, IEEE, Feb. 2010, pp. 108–109, ISBN: 978-1-4244-6033-5. DOI: 10.1109/ISSCC.2010.5434077.

[19] P. Li, J. L. Shin, G. Konstadinidis, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. David Lin, P. Loewenstein, H. Park, V. Srinivasan, D. Huang, C. Hwang, W. Hsu, and C. McAllister, "4.2 A 20nm 32-Core 64MB L3 cache SPARC M7 processor," in *2015 IEEE INTERNATIONAL SOLID-STATE CIRCUITS CONFERENCE - (ISSCC) DIGEST OF TECHNICAL PAPERS*, IEEE, Feb. 2015, pp. 1–3, ISBN: 978-1-4799-6223-5. DOI: 10.1109/ISSCC.2015.7062931.

[20] H. Sutter, "Modern C++: What You Need to Know," in *BUILD*, San Francisco, CA, 2014.

[21] Intel Corporation, *Intel ® Itanium ® Architecture Software Developer's Manual*, 2.3. 2010.

[22] D. Koufaty and D. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, Mar. 2003, ISSN: 0272-1732. DOI: 10.1109/MM.2003.1196115.

[23] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Fourth. Elsevier, Inc., 2009.

[24] T. Austin, D. Blaauw, T. Mudge, K. Flautner, M. Irwin, M. Kandemir, and V. Narayanan, "Leakage current: Moore's law meets static power," English, *Computer*, vol. 36, no. 12, pp. 68–75, Dec. 2003, ISSN: 0018-9162. DOI: 10.1109/MC.2003.1250885.

[25] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," *IEEE Micro*, vol. 32, no. 3, pp. 122–134, May 2012, ISSN: 0272-1732. DOI: 10.1109/MM.2012.17.

[26] A. A. Chien and V. Karamcheti, "Moore's Law: The First Ending and a New Beginning," *Computer*, vol. 46, no. 12, pp. 48–53, Dec. 2013, ISSN: 0018-9162. DOI: 10.1109/MC.2013.431.

[27] L. B. Kish, "End of Moore's law: thermal (noise) death of integration in micro and nano electronics," *Physics Letters A*, vol. 305, no. 3-4, pp. 144–149, Dec. 2002, ISSN: 03759601. DOI: 10.1016/S0375-9601(02)01365-8.

[28] L. Eeckhout, "Heterogeneity in Response to the Power Wall," *IEEE Micro*, vol. 35, no. 4, pp. 2–3, Jul. 2015, ISSN: 0272-1732. DOI: 10.1109/MM.2015.86.

[29] ARM Ltd. (2015). big.LITTLE Technology, [Online]. Available: https://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[30] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *PROCEEDINGS OF THE APRIL 18-20, 1967, SPRING JOINT COMPUTER CONFERENCE ON - AFIPS '67 (SPRING)*, New York, New York, USA: ACM Press, Apr. 1967, p. 483. DOI: 10.1145/1465482.1465560.

[31] J. L. Gustafson, "Reevaluating Amdahl's law," *Communications of the ACM*, vol. 31, no. 5, pp. 532–533, May 1988, ISSN: 00010782. DOI: 10.1145/42411.42415.

[32] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," *Computer*, vol. 41, no. 7, pp. 33–38, Jul. 2008, ISSN: 0018-9162. DOI: 10.1109/MC.2008.209.

[33] C. Hewitt, P. Bishop, and R. Steiger, "A universal modular ACTOR formalism for artificial intelligence," in *PROCEEDINGS OF THE 3RD INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE (IJCAI'73)*, Morgan Kaufmann Publishers Inc., Aug. 1973, pp. 235–245.

[34] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978, ISSN: 00010782. DOI: 10.1145/359576.359585.

[35] S. B. Furber, D. R. Lester, L. A. Plana, J. D. Garside, E. Painkras, S. Temple, and A. D. Brown, "Overview of the SpiNNaker System Architecture," *IEEE Transactions on Computers*, pp. 1–14, 2012, ISSN: 0018-9340. DOI: 10.1109/TC.2012.142.

[36] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sep. 1972, ISSN: 0018-9340. DOI: 10.1109/TC.1972.5009071.

[37] R. Duncan, "A survey of parallel computer architectures," *Computer*, vol. 23, no. 2, pp. 5–16, Feb. 1990, ISSN: 0018-9162. DOI: 10.1109/2.44900.

[38] J. McCutchan. (2013). Using SIMD in Dart, [Online]. Available: https://www.dartlang.org/articles/simd/ (visited on 02/26/2016).

[39] R. Haring, M. Ohmacht, T. Fox, M. Gschwind, D. Satterfield, K. Sugavanam, P. Coteus, P. Heidelberger, M. Blumrich, R. Wisniewski, A. Gara, G. Chiu, P. Boyle, N. Chist, and C. Kim, "The IBM Blue Gene/Q Compute Chip," English, *IEEE Micro*, vol. 32, no. 2, pp. 48–60, Mar. 2012, ISSN: 0272-1732. DOI: 10.1109/MM.2011.108.

[40] D. E. Shaw, J. C. Chao, M. P. Eastwood, J. Gagliardo, J. P. Grossman, C. R. Ho, D. J. Lerardi, I. Kolossváry, J. L. Klepeis, T. Layman, C. McLeavey, M. M. Deneroff, M. A. Moraes, R. Mueller, E. C. Priest, Y. Shan, J. Spengler, M. Theobald, B. Towles, S. C. Wang, R. O. Dror, J. S. Kuskin, R. H. Larson, J. K. Salmon, C. Young, B. Batson, and K. J. Bowers, "Anton, a special-purpose machine for molecular dynamics simulation," *Communications of the ACM*, vol. 51, no. 7, pp. 91–97, Jul. 2008, ISSN: 00010782. DOI: 10.1145/1364782.1364802.

[41] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z*. 2015.

[42] L. Chen, P. Jiang, and G. Agrawal, "Exploiting Recent SIMD Architectural Advances for Irregular Applications," in *14TH ANNUAL IEEE/ACM INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION (CGO 2016)*, Barcelona, Spain, 2016.

[43] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD Vectorization for In-Memory Databases," in *PROCEEDINGS OF THE 2015 ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA - SIGMOD '15*, New York, New York, USA: ACM Press, May 2015, pp. 1493–1508, ISBN: 9781450327589. DOI: 10.1145/2723372.2747645.

[44] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. Snove, and O. Birkeland, "A recursive MISD architecture for pattern matching," English, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 12, no. 7, pp. 727–734, Jul. 2004, ISSN: 1063-8210. DOI: 10.1109/TVLSI.2004.830918.

[45] B.-O. Schneider and J. Rossignac, "M-Buffer: A flexible MISD architecture for advanced graphics," *Computers & Graphics*, vol. 19, no. 2, pp. 239–246, Mar. 1995, ISSN: 00978493. DOI: 10.1016/0097-8493(94)00149-S.

[46] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," English, in *2012 INNOVATIVE PARALLEL COMPUTING (INPAR)*, IEEE, May 2012, pp. 1–13, ISBN: 978-1-4673-2633-9. DOI: 10.1109/InPar.2012.6339601.

[47] D.-H. Lee, H. Moon, J. C. Koo, and H. R. Choi, "MATRIX: A message passing interface for MPMD (Multiple Program Multiple Data) applications on heterogeneous distributed network," English, in *2012 9TH INTERNATIONAL CONFERENCE ON UBIQUITOUS ROBOTS AND AMBIENT INTELLIGENCE (URAI)*, IEEE, Nov. 2012, pp. 310–315, ISBN: 978-1-4673-3112-8. DOI: 10.1109/URAI.2012.6463002.

[48] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program GPUs for general-purpose uses," *ACM SIGPLAN Notices*, vol. 41, no. 11, pp. 325–335, Oct. 2006, ISSN: 03621340. DOI: 10.1145/1168918.1168898.

[49] E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Communications of the ACM*, vol. 8, no. 9, p. 569, Sep. 1965, ISSN: 00010782. DOI: 10.1145/365559.365617.

[50] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communications of the ACM*, vol. 17, no. 8, pp. 453–455, Aug. 1974, ISSN: 00010782. DOI: 10.1145/361082.361093.

[51] M. J. Quinn, *Parallel Computing: Theory and Practice*. McGraw-Hill, Inc., 1994, ISBN: 0-07-051294-9.

[52] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *PROCEEDINGS OF THE TENTH ACM SIGPLAN SYMPOSIUM ON PRINCIPLES AND PRACTICE OF PARALLEL PROGRAMMING - PPOPP '05*, New York, New York, USA: ACM Press, Jun. 2005, pp. 48–60, ISBN: 1595930809. DOI: 10.1145/1065944.1065952.

[53] J. Hill and D. Skillicorn, "Practical barrier synchronisation," English, in *PROCEEDINGS OF THE SIXTH EUROMICRO WORKSHOP ON PARALLEL AND DISTRIBUTED PROCESSING - PDP '98* -, Madrid: IEEE Comput. Soc, 1998, pp. 438–444, ISBN: 0-8186-8332-5. DOI: 10.1109/EMPDP.1998.647231.

[54] D. Chen, N. Eisley, P. Heidelberger, R. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and J. Parker, "The IBM Blue Gene/Q Interconnection Fabric," English, *IEEE Micro*, vol. 32, no. 1, pp. 32–43, Jan. 2012, ISSN: 0272-1732. DOI: 10.1109/MM.2011.96.

[55] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*. The Benjamin/Cummings Publishing Company, Inc., 1994, ISBN: 0-8053-3170-0.

[56] ARM Ltd. (2015). CoreLink CCN-502, [Online]. Available: https://www.arm.com/products/system-ip/interconnect/corelink-ccn-502.php (visited on 02/29/2016).

[57] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *DESIGN AUTOMATION CONFERENCE, 2001. PROCEEDINGS*, 2001, pp. 684–689. DOI: 10.1109/DAC.2001.156225.

[58] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Elsevier, Inc., 2004.

[59] S. Satpathy, K. Sewell, T. Manville, Y.-p. Chen, R. Dreslinski, D. Sylvester, T. Mudge, and D. Blaauw, "A 4.5Tb/s 3.4Tb/s/W 64×64 switch fabric with self-updating least-recently-granted priority and quality-of-service arbitration in 45nm CMOS," in *SOLID-STATE CIRCUITS CONFERENCE DIGEST OF TECHNICAL PAPERS (ISSCC), 2012 IEEE INTERNATIONAL*, IEEE, Feb. 2012, pp. 478–480, ISBN: 978-1-4673-0377-4. DOI: 10.1109/ISSCC.2012.6177098.

[60] Intel Corporation, *An Introduction to the Intel® QuickPath Interconnect*. 2009.

[61] HyperTransport™ Consortium, *HyperTransport™ I/O Technology Overview—An Optimized, Low-latency Board-level Architecture*. 2004.

[62] D. Chen, N. Eisley, P. Heidelberger, R. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. Satterfield, B. Steinmacher-Burow, and 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*, Seatle, WA, 2011, pp. 1–10.

[63] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su, "Myrinet–a gigabit-per-second local-area network," *Symposium Record Hot Interconnects II*, no. February, pp. 161–180, 1995. DOI: 10.1109/CONECT.1994.765346.

[64] P. Geoffray, "OPIOM Off-Processor IO with Myrinet," *Parallel Processing Letters*, vol. 11, no. 2-3, pp. 237–250, Sep. 2001, ISSN: 01296264. DOI: 10.1016/S0129-6264(01)00055-5.

[65] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick, "An evaluation of current high-performance networks," in *PROCEEDINGS INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM*, IEEE Comput. Soc, 2003, p. 10, ISBN: 0-7695-1926-1. DOI: 10.1109/IPDPS.2003.1213106.

[66] S. Majumder and S. Rixner, "Comparing Ethernet and Myrinet for MPI communication," in *PROCEEDINGS OF THE 7TH WORKSHOP ON WORKSHOP ON LANGUAGES, COMPILERS, AND RUN-TIME SUPPORT FOR SCALABLE SYSTEMS - LCR '04*, New York, New York, USA: ACM Press, 2004, pp. 1–7. DOI: 10.1145/1066650.1066659.

[67] M. J. Rashti and A. Afsahi, "10-Gigabit iWARP Ethernet: Comparative Performance Analysis with InfiniBand and Myrinet-10G," in *2007 IEEE INTERNATIONAL PARALLEL AND DISTRIBUTED PROCESSING SYMPOSIUM*, IEEE, 2007, pp. 1–8, ISBN: 1-4244-0909-8. DOI: 10.1109/IPDPS.2007.370480.

[68] (2016). Node.js, [Online]. Available: https://nodejs.org/en/ (visited on 03/01/2016).

[69] (2014). File System, [Online]. Available: https://nodejs.org/docs/latest-v0.12.x/api/fs.html (visited on 03/01/2016).

[70] (2016). Asynchronous I/O, event loop, coroutines and tasks, [Online]. Available: https://docs.python.org/3/library/asyncio.html (visited on 03/01/2016).

[71] (). pthreads(7): POSIX threads, [Online]. Available: http://linux.die.net/man/7/pthreads (visited on 03/01/2016).

[72] (2016). std::thread, [Online]. Available: http://en.cppreference.com/w/cpp/thread/thread (visited on 03/01/2016).

[73] (2015). std::promise, [Online]. Available: http://en.cppreference.com/w/cpp/thread/promise (visited on 03/01/2016).

[74] (2015). std::future, [Online]. Available: http://en.cppreference.com/w/cpp/thread/future (visited on 03/01/2016).

[75] (2015). std::packaged_task, [Online]. Available: http://en.cppreference.com/w/cpp/thread/packaged%7B%5C_%7Dtask (visited on 03/01/2016).

[76] (2016). Asynchronous Programming with Async and Await, [Online]. Available: https://msdn.microsoft.com/en-gb/library/hh191443.aspx (visited on 03/01/2016).

[77] B. Chapman, G. Jost, and R. van der Pas, *Using OpenMP : Portable Shared Memory Parallel Programming*. Cambridge, MA, USA: MIT Press, 2007.

[78] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using Advanced MPI:Modern Features of the Message-Passing Interface*. MIT Press, 2014.

[79] F. Akgul, *ZeroMQ*. Birmingham, UK: Packt Publishing Ltd., 2013, ISBN: 9781782161059.

[80] A. N. Burkitt, "A review of the integrate-and-fire neuron model: I. Homogeneous synaptic input.," *Biological cybernetics*, vol. 95, no. 1, pp. 1–19, Jul. 2006, ISSN: 0340-1200. DOI: 10.1007/s00422-006-0068-6.

[81] E. M. Izhikevich and G. M. Edelman, "Large-scale model of mammalian thalamocortical systems.," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 105, no. 9, pp. 3593–8, Mar. 2008, ISSN: 1091-6490. DOI: 10.1073/pnas.0712231105.

[82] X. Jin, A. Rast, F. Galluppi, S. Davies, and S. Furber, "Implementing spike-timing-dependent plasticity on SpiNNaker neuromorphic hardware," in *THE 2010 INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS (IJCNN)*, IEEE, Jul. 2010, pp. 1–8, ISBN: 978-1-4244-6916-1. DOI: 10.1109/IJCNN.2010.5596372.

[83] E. M. Izhikevich, "Polychronization: computation with spikes.," *Neural computation*, vol. 18, no. 2, pp. 245–82, Feb. 2006, ISSN: 0899-7667. DOI: 10.1162/089976606775093882.

[84] (2015). BrainScaleS, [Online]. Available: https://brainscales.kip.uni-heidelberg.de/ (visited on 03/02/2016).

[85] J. Schemmel, D. Briiderle, A. Griibl, M. Hock, K. Meier, and S. Millner, "A wafer-scale neuromorphic hardware system for large-scale neural modeling," in *PROCEEDINGS OF 2010 IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS*, IEEE, May 2010, pp. 1947–1950, ISBN: 978-1-4244-5308-5. DOI: 10.1109/ISCAS.2010.5536970.

[86] S. Scholze, S. Schiefer, J. Partzsch, S. Hartmann, C. G. Mayr, S. Höppner, H. Eisenreich, S. Henker, B. Vogginger, and R. Schüffny, "VLSI Implementation of a 2.8 Gevent/s Packet-Based AER Interface with Routing and Event Sorting Functionality.," English, *Frontiers in neuroscience*, vol. 5, p. 117, Jan. 2011, ISSN: 1662-453X. DOI: 10.3389/fnins.2011.00117.

[87] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," en, *Science*, vol. 345, no. 6197, pp. 668–673, Aug. 2014, ISSN: 0036-8075. DOI: 10.1126/science.1254642.

[88] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G.-J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, "TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 34, no. 10, pp. 1537–1557, Oct. 2015, ISSN: 0278-0070. DOI: 10.1109/TCAD.2015.2474396.

[89] S. W. Moore, P. J. Fox, S. J. Marsh, A. T. Markettos, and A. Mujumdar, "Bluehive - A Field-Programable Custom Computing Machine for Extreme-Scale Real-Time Neural Network Simulation," in *2012 IEEE 20TH INTERNATIONAL SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES*, IEEE, Apr. 2012, pp. 133–140.

[90] L. a. Plana, S. B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang, "A GALS Infrastructure for a Massively Parallel Multiprocessor," *IEEE Design & Test of Computers*, vol. 24, no. 5, pp. 454–463, Sep. 2007, ISSN: 0740-7475. DOI: 10.1109/MDT.2007.149.

[91] J. Wu and S. Furber, "Delay Insensitive Chip-to-Chip Interconnect Using Incomplete 2-of-7 NRZ Data Encoding," in *PROCEEDINGS OF THE 18TH UK ASYNCHRONOUS FORUM*, Newcastle upon Tyne, UK: University of Newcastle, 2006, pp. 16–19.

[92] N. T. Carnevale and M. L. Hines. (2013). NEURON - for empirically-based simulations of neurons and networks of neurons, [Online]. Available: https://www.neuron.yale.edu/neuron/ (visited on 03/03/2016).

[93] M. Hines, "The NEURON Simulation Program," in *THE KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE*, ser. The Kluwer International Series in Engineering and Computer Science, J. Skrzypek, Ed., vol. 254, Boston, MA: Springer US, 1994, ch. Neural Net, pp. 147–163, ISBN: 978-1-4613-6180-0. DOI: 10.1007/978-1-4615-2736-7.

[94] M. L. Hines and N. T. Carnevale, "The NEURON Simulation Environment," en, *Neural Computation*, vol. 9, no. 6, pp. 1179–1209, Aug. 1997, ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.6.1179.

[95] H. Markram, "The blue brain project.," *Nature Reviews Neuroscience*, vol. 7, no. 2, pp. 153–160, Feb. 2006, ISSN: 1471-003X. DOI: 10.1038/nrn1848.

[96] (2016). NEST Simulator - The Neural Simulation Tool, [Online]. Available: http://www.nest-simulator.org/ (visited on 03/03/2016).

[97] M.-O. Gewaltig and M. Diesmann, "NEST (NEural Simulation Tool)," *Scholarpedia*, vol. 2, no. 4, p. 1430, Apr. 2007, ISSN: 1941-6016. DOI: 10.4249/scholarpedia.1430.

[98] H. E. Plesser, J. M. Eppler, A. Morrison, M. Diesmann, and M.-O. Gewaltig, "Efficient Parallel Simulation of Large-Scale Neuronal Networks on Clusters of Multiprocessor Computers," in *EURO-PAR 2007 PARALLEL PROCESSING: 13TH INTERNATIONAL EURO-PAR CONFERENCE, RENNES ,FRANCE , AUGUST 28-31, 2007. PROCEEDINGS*, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 672–681, ISBN: 978-3-540-74466-5. DOI: 10.1007/978-3-540-74466-5_71.

[99] R. Brette, D. Goodman, and M. Stimberg. (2016). The Brian spiking neural network simulator, (visited on 03/03/2016).

[100] D. F. M. Goodman and R. Brette, "The Brian simulator," English, *Frontiers in Neuroscience*, vol. 3, no. 2, pp. 192–197, Sep. 2009, ISSN: 1662-453X. DOI: 10.3389/neuro.01.026.2009.

[101] (). PCSIM: A Parallel neural Circuit SIMulator, [Online]. Available: http://www.lsm.tugraz.at/pcsim/ (visited on 03/03/2016).

[102] D. Pecevski, T. Natschläger, and K. Schuch, "PCSIM: a parallel simulation environment for neural circuits fully integrated with Python," English, *Frontiers in Neuroinformatics*, vol. 3, pp. 1–15, Jan. 2009, ISSN: 1662-5196. DOI: 10.3389/neuro.11.011.2009.

[103] (2015). Open Source Brain, [Online]. Available: http://www.opensourcebrain.org/ (visited on 03/03/2016).

[104] P. Gleeson, V. Steuber, and R. A. Silver, "neuroConstruct: A Tool for Modeling Networks of Neurons in 3D Space," *Neuron*, vol. 54, no. 2, pp. 219–35, Apr. 2007, ISSN: 0896-6273. DOI: 10.1016/j.neuron.2007.03.025.

[105] (2013). PyNN, [Online]. Available: http://neuralensemble.org/PyNN/ (visited on 03/03/2016).

[106] D. Brüderle, E. Müller, A. Davison, E. Muller, J. Schemmel, and K. Meier, "Establishing a novel modeling tool: a python-based interface for a neuromorphic hardware system," English, *Frontiers in neuroinformatics*, vol. 3, p. 17, Jan. 2009, ISSN: 1662-5196. DOI: 10.3389/neuro.11.017.2009.

[107] F. Galluppi, A. Rast, S. Davies, and S. Furber, "A General-Purpose Model Translation System for a Universal Neural Chip," in *LECTURE NOTES IN COMPUTER SCIENCE*, ser. Lecture Notes in Computer Science, K. W. Wong, B. S. U. Mendis, and A. Bouzerdoum, Eds., vol. 6443, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, ch. Neural Inf, pp. 58–65, ISBN: 978-3-642-17536-7. DOI: 10.1007/978-3-642-17537-4.

[108] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "PyNN: A Common Interface for Neuronal Network Simulators," English, *Frontiers in neuroinformatics*, vol. 2, p. 11, Jan. 2008, ISSN: 1662-5196. DOI: 10.3389/neuro.11.011.2008.

[109] (2015). NeuroML, [Online]. Available: https://www.neuroml.org/ (visited on 03/03/2016).

[110] (2013). NineML, [Online]. Available: http://software.incf.org/software/nineml (visited on 03/03/2016).

[111] (). Introduction to NineML, [Online]. Available: http://software.incf.org/software/nineml/wiki/introduction-to-nineml (visited on 03/03/2016).

[112] (). LEMS - Low Entropy Model Specification, [Online]. Available: http://lems.github.io/LEMS/ (visited on 03/03/2016).

[113] M. Vella, R. C. Cannon, S. Crook, A. P. Davison, G. Ganapathy, H. P. C. Robinson, R. A. Silver, and P. Gleeson, "libNeuroML and PyLEMS: using Python to combine procedural and declarative modeling approaches in computational neuroscience.," English, *Frontiers in Neuroinformatics*, vol. 8, p. 38, Jan. 2014, ISSN: 1662-5196. DOI: 10.3389/fninf.2014.00038.

[114] R. C. Cannon, P. Gleeson, S. Crook, G. Ganapathy, B. Marin, E. Piasini, and R. A. Silver, "LEMS: a language for expressing complex biological models in concise and hierarchical form and its use in underpinning NeuroML 2.," English, *Frontiers in Neuroinformatics*, vol. 8, p. 79, Jan. 2014, ISSN: 1662-5196. DOI: 10.3389/fninf.2014.00079.

[115] University of Manchester, *SpiNNaker datasheet version 2.02*, 2011.

[116] E. Painkras, L. A. Plana, J. Garside, S. Temple, F. Galluppi, C. Patterson, D. R. Lester, A. D. Brown, and S. B. Furber, "SpiNNaker: A 1-W 18-Core System-on-Chip for Massively-Parallel Neural Network Simulation," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 8, pp. 1943–1953, Aug. 2013, ISSN: 0018-9200. DOI: 10.1109/JSSC.2013.2259038.

[117] S. Furber and A. Brown, "Biologically-Inspired Massively-Parallel Architectures - Computing Beyond a Million Processors," in *2009 NINTH INTERNATIONAL CONFERENCE ON APPLICATION OF CONCURRENCY TO SYSTEM DESIGN*, IEEE, Jul. 2009, pp. 3–12, ISBN: 978-0-7695-3697-2. DOI: 10.1109/ACSD.2009.17.

[118] S. Furber and J. Bainbridge, "Future Trends in SoC Interconnect," in *2005 International Symposium on System-on-Chip*, IEEE, 2005, pp. 183–186, ISBN: 0-7803-9294-9. DOI: 10.1109/ISSOC.2005.1595673.

[119] J. Bainbridge and S. Furber, "Chain: a delay-insensitive chip area interconnect," *IEEE Micro*, vol. 22, no. 5, pp. 16–23, Sep. 2002, ISSN: 0272-1732. DOI: 10.1109/MM.2002.1044296.

[120] W. Bainbridge, W. Toms, D. Edwards, and S. Furber, "Delay-insensitive, point-to-point interconnect using m-of-n codes," in *NINTH INTERNATIONAL SYMPOSIUM ON ASYNCHRONOUS CIRCUITS AND SYSTEMS, 2003. PROCEEDINGS.*, IEEE Comput. Soc, 2003, pp. 132–140, ISBN: 0-7695-1898-2. DOI: 10.1109/ASYNC.2003.1199173.

[121] J. Wu, S. Furber, and J. Garside, "A Programmable Adaptive Router for a GALS Parallel System," in *2009 15TH IEEE SYMPOSIUM ON ASYNCHRONOUS CIRCUITS AND SYSTEMS*, IEEE, May 2009, pp. 23–31, ISBN: 978-0-7695-3616-3. DOI: 10.1109/ASYNC.2009.17.

[122] K. Boahen, "Point-to-point connectivity between neuromorphic chips using address events," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 47, no. 5, pp. 416–434, May 2000, ISSN: 10577130. DOI: 10.1109/82.842110.

[123] K. Dugan, A. Brown, and J. Reeve, "SpiNNaker Peripheral IO," University of Southampton, Southampton, Tech. Rep., 2013. [Online]. Available: http://spinnaker.ecs.soton.ac.uk/docs/spin-io/pdfs/SpiNNakerPeripheralIO%5C_latest.pdf.

[124] S. Temple, *AppNote 10 - SpiNN-24B - Brief Guide*, Private Correspondence, 2015.

[125] T. Sharp, C. Patterson, and S. Furber, "Distributed configuration of massively-parallel simulation on SpiNNaker neuromorphic hardware," in *THE 2011 INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS*, IEEE, Jul. 2011, pp. 1099–1105, ISBN: 978-1-4244-9635-8. DOI: 10.1109/IJCNN.2011.6033346.

[126] M. M. Khan, J. Navaridas, A. D. Rast, X. Jin, L. A. Plana, M. Lujan, J. V. Woods, J. Miguel-Alonso, and S. B. Furber, "Event-Driven Configuration of a Neural Network CMP System over a Homogeneous Interconnect Fabric," in *2009 EIGHTH INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED COMPUTING*, IEEE, Jun. 2009, pp. 54–61, ISBN: 978-0-7695-3680-4. DOI: 10.1109/ISPDC.2009.25.

[127] K. Dugan, "SpiNNaker Uploader Documentation," University of Southampton, Southampton, Tech. Rep., 2013. [Online]. Available: http://spinnaker.ecs.soton.ac.uk/docs/uploader/pdfs/SpiNNakerUploader%7B%5C_%7Dlatest.pdf.

[128] A. Brown, D. Lester, and L. Plana, "SpiNNaker: The design automation problem," *Advances in Neuro-Information Processing*, vol. 5507, pp. 1049–1056, 2009. DOI: 10.1007/978-3-642-03040-6_127.

[129] A. D. Rast, M. Khan, and S. B. Furber, "Virtual synaptic interconnect using an asynchronous network-on-chip," in *2008 IEEE INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS (IEEE WORLD CONGRESS ON COMPUTATIONAL INTELLIGENCE)*, IEEE, Jun. 2008, pp. 2727–2734, ISBN: 978-1-4244-1820-6. DOI: 10.1109/IJCNN.2008.4634181.

[130] A. D. Rast, L. A. Plana, S. R. Welbourne, and S. Furber, "Event-driven MLP implementation on neuromimetic hardware," in *THE 2012 INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS (IJCNN)*, IEEE, Jun. 2012, pp. 1–8, ISBN: 978-1-4673-1490-9. DOI: 10.1109/IJCNN.2012.6252821.

[131] A. D. Rast, F. Galluppi, X. Jin, and S. Furber, "The Leaky Integrate-and-Fire neuron: A platform for synaptic model exploration on the SpiNNaker chip," in *THE 2010 INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS (IJCNN)*, IEEE, Jul. 2010, pp. 1–8, ISBN: 978-1-4244-6916-1. DOI: 10.1109/IJCNN.2010.5596364.

[132] M. Lujan, L. A. Plana, S. Davies, S. Temple, and S. B. Furber, "Modeling Spiking Neural Networks on SpiNNaker," *Computing in Science & Engineering*, vol. 12, no. 5, pp. 91–97, Sep. 2010, ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.112.

[133] S. B. Furber and J. V. Woods, "Efficient modelling of spiking neural networks on a scalable chip multiprocessor," in *2008 IEEE INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS (IEEE WORLD CONGRESS ON COMPUTATIONAL INTELLIGENCE)*, IEEE, Jun. 2008, pp. 2812–2819, ISBN: 978-1-4244-1820-6. DOI: 10.1109/IJCNN.2008.4634194.

[134] B. K. Choi and D. Kang, *Modeling and Simulation of Discrete Event Systems*. John Wiley & Sons, 2013, ISBN: 9781118732854.

[135] J. Banks, J. S. Carson, and B. L. Nelson, *Discrete-Event System Simulation: Pearson New International Edition*. Pearson, 2013.

[136] X. Jin, F. Galluppi, C. Patterson, A. Rast, S. Davies, S. Temple, and S. Furber, "Algorithm and software for simulation of spiking neural networks on the multi-chip SpiNNaker system," in *THE 2010 INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORKS (IJCNN)*, IEEE, Jul. 2010, pp. 1–8, ISBN: 978-1-4244-6916-1. DOI: 10.1109/IJCNN.2010.5596759.

[137] K. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, Sep. 1979, ISSN: 0098-5589. DOI: 10.1109/TSE.1979.230182.

[138] C. Bai, "Parallel discrete event simulation on the SpiNNaker engine," Ph.D. Thesis, University of Southampton, 2013.

[139] ARM Ltd. (2010). RealView Development Suite, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.rvds/index.html (visited on 03/21/2016).

[140] D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, 1997, ISBN: 1886529019.

[141] K. W. Morton and D. F. Mayers, *Numerical Solution of Partial Differential Equations: An Introduction*, 2nd. Cambridge, UK: Cambridge University Press, 2005, ISBN: 0-521-60793-0.

[142] L. D. Solano-Quinde and B. M. Bode, "Module Prototype for Online Failure Prediction for the IBM Blue Gene/L," in *2008 IEEE INTERNATIONAL CONFERENCE ON ELECTRO/INFORMATION TECHNOLOGY*, IEEE, May 2008, pp. 470–474, ISBN: 978-1-4244-2029-2. DOI: 10.1109/EIT.2008.4554349.

[143] J. Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC v8 architecture," in *PROCEEDINGS INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS*, IEEE Comput. Soc, 2002, pp. 409–415, ISBN: 0-7695-1597-5. DOI: 10.1109/DSN.2002.1028926.

[144] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, ISSN: 00010782. DOI: 10.1145/1327452.1327492.

[145] ARM Ltd. (2011). Using Scatter-loading Description Files, [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.kui0101a/armlink_babddhbf.htm (visited on 03/24/2016).

[146] ——, (2010). ARM Compiler toolchain Compiler Reference—function attribute, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/Cacbgief.html (visited on 03/24/2016).

[147] ——, (2010). ARM Compiler toolchain Compiler Reference—variable attribute, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/Caccache.html (visited on 03/24/2016).

[148] ——, (2010). ARM Compiler toolchain Compiler Reference—#pragma arm section, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0491c/BCFJBABB.html (visited on 03/24/2016).

[149] ——, (2010). Importing linker-defined symbols in ARM assembler, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0474c/CHDCGJFI.html (visited on 03/24/2016).

[150] ——, (2010). Importing linker-defined symbols in C and C++, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0474c/CHDBIJJD.html (visited on 03/24/2016).

[151] ——, (2006). ARM968E-S™ Technical Reference Manual—CP15 c7 core control operations, [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0311d/I1014521.html (visited on 03/24/2016).