Chapter 4

Stacks of Software Stacks

By Andrew Rowley, Oliver Rhodes, Petruț Bogdan, Christian Brenninkmeijer, Simon Davidson, Donal Fellows, Steve Furber, Andrew Gait, Michael Hopkins, David Lester, Mantas Mikaitis, Luis Plana, and Alan Stokes

DOI: 10.1561/9781680836530.ch4

Published in *SpiNNaker – A Spiking Neural Network Architecture* by S. Furber and P. Bogdan (eds.). 2020. ISBN 978-1-68083-596-0. E-ISBN 978-1-68083-653-0.

Suggested citation: Andrew Rowley *et al.* 2020. "Stacks of Software Stacks" in *SpiNNaker – A Spiking Neural Network Architecture*. Edited by S. Furber and P. Bogdan. pp. 79–128. Now Publishers. DOI: 10.1561/9781680836530.ch4.

now
the essence of knowledge

All hope abandon, ye who enter here.

— Dante Alighieri, Inferno

Alongside the job of designing and producing the hardware, there is the equally challenging task of constructing software that allows users to exploit the capabilities of the machine. Using a large parallel computing system such as SpiNNaker often requires expert knowledge to be able to create and debug code that is designed to be executed in a distributed and parallel fashion. More recently, software stacks have been created which try to abstract this process away from the end user by the use of explicit interfaces or by defining the problem in a form which is easier to map into a distributed system. In this chapter, we describe the SpiNNaker software stacks upon which most of the applications described in subsequent chapters are supported. It is built by merging slightly modified versions of the work presented by Rowley *et al.* [213], covering the software tools that allow the running of generic applications – the SpiNNaker Tools (SpiNNTools); and Rhodes *et al.* [207], covering the tools that specifically support the simulation of Spiking Neural Networks (SNNs) – the SpiNNaker backend for PyNN (sPyNNaker).

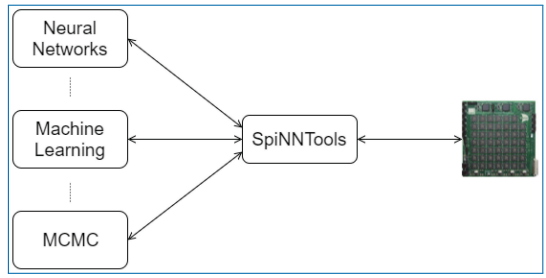

Figure 4.1. Applications using SpiNNTools to control SpiNNaker.

4.1 Introduction

A growing number of users are now using SpiNNaker for a wide range of tasks, including Computational Neuroscience [3] and Neurorobotics [1, 48, 209] for which the platform was originally designed, but also machine learning [240], and general parallel computation tasks, such as Markov Chain Monte Carlo inference computations [161]. The provision of a software stack for this platform aims to provide a base for the various applications, making it easier for them to exploit the full potential of the platform. Additionally, users will gain the advantage of any improvement in the underlying tools without requiring changes to their software (or at most only minor interface changes should they be required). A basic overview of this approach is seen in Figure 4.1.

The software stack allows the user to describe their computational requirements in the form of a graph, where the vertices represent the units of computation, and the edges represent the communication of data between the computational units. This graph is described in a high-level language and the software then maps this directly onto an available SpiNNaker machine. The SpiNNaker platform as a whole is intended to improve the overall execution time of the computational problems mapped onto it, and so the time taken to execute this mapping is critical; if it takes too long, it will dwarf the computational execution time of the problem itself.

The problem of writing code to run on the cores of the SpiNNaker machine is discussed in more detail by Brown *et al.* [25], along with the types of applications which might be suitable to execute on the platform. The software assumes that the application has already been designed to run in parallel on the platform; the SpiNNTools software then works to map that parallel application onto the machine, execute it and extract any results, along with any relevant data about the machine.

4.2 Making Use of the SpiNNaker Architecture

The nature of the SpiNNaker chip has important implications for the software running on the system. This section is a short recap of Chapters 2 and 3. Firstly, it must be possible to break up the computation of the application into units small enough that the code for each part fits on a single core. The SDRAM is shared between the cores on a single chip, and this property can be used by the application to allow cores to operate on the same data within the same chip. A small amount of data can be shared with cores operating on other chips as well through communication via the SpiNNaker router. The SpiNNaker boards can be connected together to form an even larger grid of chips, so appropriately parallelisable software could potentially be scaled to run on up to 1 million cores.

The SpiNNaker router is initially set up to handle the routeing of system-level data. The data to be sent by applications make use of the multicast packet type, meaning that a packet sent from a single source can be routed to multiple destinations simultaneously. To make multicast routeing work, the routeing tables of the router must be set up; this process is described in Section 4.7.

Each chip has an Ethernet controller, although in practice only one chip is connected to the Ethernet connector on each board. The chip with the Ethernet connected to it is then called the Ethernet chip, and this is used to communicate with the outside world, allowing, for example, the loading of data and applications. Communications with other chips on a board from outside of the machine must therefore go via the Ethernet chip; system-level packets are used to effect this communication between chips. In practice, the Ethernet connector of every board in a SpiNNaker machine is connected and configured, although this is not a requirement.

SpiNNaker machines are designed to be fault tolerant, so it is possible to have a functional machine with some missing parts. For example, it is normal that some of the SpiNNaker chips have 17 instead of 18 working cores, and sometimes even fewer than this as operational cores are tested more thoroughly than the testing done at manufacture. Additionally, machines can have whole chips that have been found to have faults, as well as some links broken between the chips and boards. The machine includes memory onto which faults can be stored statically in a *blacklist*, so that during the boot process these parts of the machine can be hidden to avoid using them.

SpiNNaker machines can be connected to external devices through either a SpiNNaker link connector, of which there is one on every 48-node board, or a spiNNlink SATA connector, of which there are 9 on each board; of those, 6 are used

to connect to other boards. This, along with the low power requirements, makes the machine particularly useful for robotics applications, since the board can be connected directly to the robot without any need of other equipment. The only requirement is that the external devices must be configured to talk to the machine using SpiNNaker packets. The links can be configured to connect directly to a subset of the SpiNNaker chips on the board, and entries in the routeing tables of those chips can be used to send packets to any connected device and to route packets received from the devices across the SpiNNaker network.

4.3 SpiNNaker Core Software

The ARM968 cores can execute instructions from the ITCM using the ARM or Thumb instruction sets; generally, this code is generated from compiled C code using either the GNU's Not Unix (GNU) gcc compiler[1] or the ARM armcc compiler.[2] To this end, a library known as the SpiNNaker Application Runtime Kernel (SARK) has been written which allows access to the features of the SpiNNaker core and chip [25]. Additionally, software called the SpiNNaker Control And Monitor Program (SCAMP) has also been written which allows one of the cores to operate as a monitor processor through which the chip can be controlled [25], allowing, for example, the loading of compiled applications onto the other cores of the chip, the reading and writing of the SDRAM, the loading of the SpiNNaker routeing tables and, of course, controlling the operation of the chip's blinkenlight. SCAMP software can also map out parts of the machine known to be faulty when it is first loaded. Thus, when a description of the machine is obtained via SCAMP, only working parts should be present. The list of faults is stored on the boards themselves and can be updated dynamically if other parts are subsequently found to be faulty.

The SCAMP code can be loaded onto one core on every chip of the machine, and these cores then coordinate with each other allowing communication to any chip via any Ethernet connector on the machine (see below). This communication makes use of the SpiNNaker Datagram Protocol (SDP) [64], which is encapsulated into User Datagram Protocol (UDP) packets when going off machine to external devices. Communication out of the machine from any core is achieved by using Internet Protocol (IP) Tags. The SCAMP monitor processor on each Ethernet chip maintains a list of up to 8 IP Tags, which maps between values in the tag field of the

1. https://developer.arm.com/open-source/gnu-toolchain/gnu-rm/downloads
2. https://developer.arm.com/products/software-development-tools/compilers/legacy-compiler-releases

SDP packets and an external IP address and port. When a packet is received that is destined to go out via the Ethernet (identified in the SDP packet header), this table is consulted and an UDP packet is formed containing the packet and this is sent to the IP address and port given in the table. The table can also contain Reverse IP Tags, where an UDP packet received from an external source is mapped from the UDP port in the packet to a specific chip and core on the machine, where the data of the packet are extracted and put into an SDP packet before being forwarded to the given core.

SARK provides a hardware abstraction layer, simplifying interaction with the DMA, network interface and communications controllers. SpiNNaker1 API (SpiN1API) provides an event-based operating system, as shown in Figure 4.16, with three processing threads per core: one for task queuing, one for task dispatch and one to service Fast Interrupt Request (FIQ). SpiN1API also provides the mechanism to link software callbacks to hardware events and enables triggering of actions such as sending a packet to another core and initiating a DMA. Callbacks are registered with different priority levels ranging from -1 to 2 depending on their desired function, with lower numbers scheduled preferentially. Callback tasks of priority 1 and 2 can be queued (in queues of maximum length 15), with new events added to the back of the queue. Callbacks of priority -1 and 0 are not queued, but instead pre-empt tasks assigned higher priority level numbers. Operation of this system follows the flow detailed in Figure 4.16(a).

The scheduler thread places callbacks in queues for priority levels 1 and above, and the dispatcher picks these callbacks and executes them based on priority. When the dispatcher is executing a callback of priority 1 or higher, and a callback of priority 0 is scheduled, this task pre-empts that currently being executed causing it to be suspended until the higher priority callback has completed. Callbacks of priority -1 use the FIQ thread to interact with the scheduler and dispatcher, enabling fast response and pre-emption of priority 0 and above tasks. Pointers are stored allowing fast access to the callback code, and the processor switches to FIQ banked registers to avoid the need for stacking [230], optimising the response time of priority -1 callbacks. However, this optimised performance limits the application to registering only a single -1 priority event and callback.

4.4 Booting a Million Core Machine

The process of booting the machine is shown in Figure 4.2. When the machine is first powered up, the cores on every chip start executing the boot ROM image. This is stored within the chip and cannot be altered. After testing the ITCM and

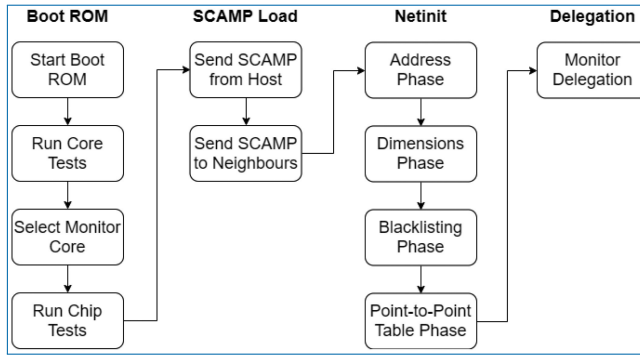

Figure 4.2. The stages of the SpiNNaker boot process.

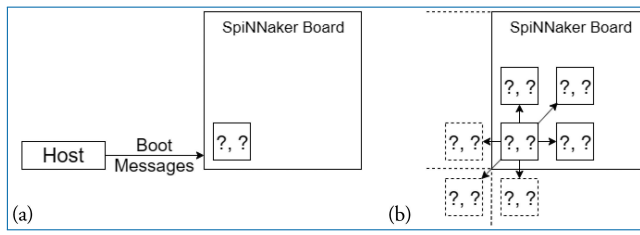

Figure 4.3. Booting SCAMP on the machine. (a) The SCAMP image is encoded in SpiNNaker boot messages and sent to the machine, where it is loaded on to the selected monitor processor of the Ethernet chip. (b) The SCAMP image is sent to neighbouring chips, which might include chips on adjacent boards, using NN packets.

DTCM of the core, the image then proceeds to determine if the core executing it is to be the monitor, through reading a mutex in the chip's *System Controller*; the first core to read this locks the mutex and so becomes the monitor. The processor selected as monitor now performs further tests on the shared parts of the chip.

Once the tests are complete, the Ethernet chips are set up to listen for boot messages being transmitted using UDP on port 54321. As shown in Figure 4.3(a), the host now sends the SCAMP image to one of these Ethernet chips; it is not critical which of these is selected, as the SCAMP software is set up to work out the dimensions of the machine and the coordinates once it has been loaded. The boot messages consist of a start command, followed by a series of 256-byte data blocks (with an appropriate header to indicate the order), followed by a completion command. If all the blocks are successfully received and assembled, the code stored in the data blocks is copied to the ITCM of the monitor processor and executed.

The current version of the SCAMP application starts with an initialisation phase where various parts of the hardware on the chip are set up for operation. The

code is then transferred to all neighbouring chips using NN packets, as shown in Figure 4.3(b). Note that at this point, SCAMP does not know how many chips are there in the whole machine, and P2P routeing tables have not been initialised, so the only protocol available for communication between the chips is nearest neighbour. To this end, SCAMP establishes a protocol to determine whether to forward NN packets received to other neighbouring chips, and down which links.

Once the image has been transferred, the core now enters the 'netinit' stage, whereby communications with all other chips on the board is established, and the point-to-point routeing tables are built. This stage proceeds as follows:

1. *Address Phase*. During this phase, each SCAMP computes and sends out its computed coordinates based on the coordinates it receives from its neighbours; for example, if it receives [0, 0] from the 'west' link, it will assume that its coordinates are [0, 1], and if it receives [0, 0] from the 'north' link, it will assume its coordinates are [−1, 0] (coordinates are allowed to be negative at this stage). This phase continues until no new coordinates are received within a given time period.
2. *Dimensions Phase*. Each SCAMP sends its perceived dimensions of the machine based on the dimensions received from its neighbours. This again continues until no change of dimensions has occurred within a given time period.
3. *Blacklisting Phase*. The blacklist is sent from the Ethernet chip of each board to the other chips on the same board. This may result in the current monitor core discovering it is blacklisted. This is noted and delegation is then set up.
4. *Point-to-Point Table Phase*. Each SCAMP sends its coordinates once again, and these are forwarded on along with a hop count, so that every chip receives them eventually. These are used to update the point-to-point tables based on the direction in which the coordinates are received, along with the hop count to allow the use of the shortest route.
5. *Monitor Delegation Phase*. If the current SCAMP core has been blacklisted, it now delegates to another core that has not. This is done at this late stage to avoid interfering with the rest of the setup process.

Note that delegation of a blacklisted monitor core will not happen until after the 'netinit' phase has completed. The monitor core tends to be selected from a subset of the cores on the chip due to manufacturing properties; this means that boards where a core which is in this subset is so broken that it cannot perform the steps up to this point will not work with this system. A possible future change would therefore be to perform the blacklisting phase earlier in the process.

4.5 Previous Software Versions

Using SpiNNaker machines in the past required end users to load compiled applications and routeing tables manually onto the SpiNNaker machine through the use of the low level *ybug* software included with the aforementioned libraries.[3] Other software was then designed to ease the development of application code for end users. These consisted of: the aforementioned low-level libraries SARK and SpiN1API, and the monitor core software SCAMP, a collection of C code which represented models known in the neuroscience community and defined by the PyNN 0.6 language [44] and a collection of Python code which translates PyNN models onto a SpiNNaker machine. These pieces of software were amalgamated into a software package known as PACMAN 48 [68] and supported the main end-user community of computational neuroscientists for a number of years. These tools had the following limitations:

- They only supported SpiNNaker machines consisting of a single SpiNN-3 or SpiNN-5 board.
- They were designed to support only the computational neuroscience community, and thus, non-neural applications were not supported.
- End users were still expected to have expertise in using the SpiNNaker hardware. This was required as they were expected to run separate scripts manually, which together and in this order:
    1. Boot the SpiNNaker machine,
    2. Load executables onto the SpiNNaker machine,
    3. Load data objects onto SpiNNaker,
    4. Check when the executing code finished,
    5. Extract data from the SpiNNaker machine.

It was decided that a new software stack should be built to address these issues. The intention of this is to support a range of suitable applications executing on the SpiNNaker hardware by providing a flexible abstraction layer where the end user represents their problem as a graph, which is then executed on the SpiNNaker machine without requiring such a low-level knowledge of how the machine works, thus overcoming the issues mentioned above. This concept is briefly mentioned as 'The Uploader' by Brown *et al.* [25], although the framework is more complete in that it also:

- allows the user to express the generation of the data structures to be loaded (and possibly reloaded when changes have been made);

3. Available from https://github.com/SpiNNakerManchester/spinnaker_tools/releases

- controls the execution flow of the application where required;
- aids in the storage and retrieval of data recorded during the execution;
- and extracts and presents provenance data which can be used to determine the correctness of the results.

4.6 Data Structures

4.6.1 SpiNNaker Machines

A SpiNNaker machine is represented as a set of Python classes as shown in Figure 4.4, with a main Machine class which then contains instances of classes for each of the parts of the machine represented. This data structure includes the important details of the machine for mapping purposes, including the chips, cores and links available, as well as the speed of each core, and the SDRAM available and the number of routeing entries available on each chip (in case some of this resource is used by the system software, as it is in the case of SCAMP). As well as internally representing a physical, real-world machine with all its faults mapped out, this representation also allows the instantiation of a virtual machine for testing in the absence of connected hardware. The virtual machine can be further modified to simulate hardware faults and analyse software behaviour.

The connection of external devices, such as a silicon retina or a motor to the machine, is represented using 'virtual chips'. A virtual chip will be given coordinates of a chip that does not exist in the physical machine and is therefore marked as virtual. The coordinates do not have to align with the rest of the machine, as the location where the chip is connected to the other real chips in the machine is also identified. This allows any algorithm to detect that virtual chips are present if

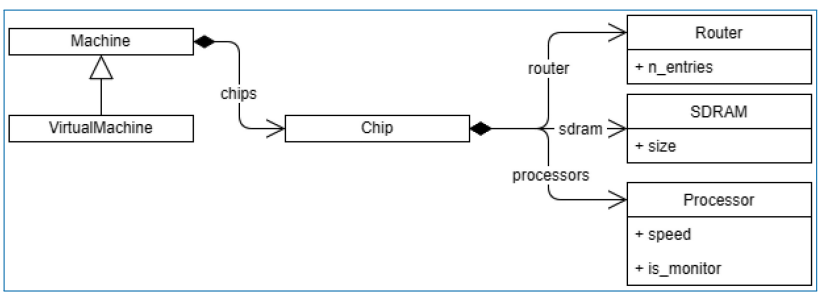

Figure 4.4. The Python class hierarchy for SpiNNaker Machine representation. The machine contains a list of chips, and each chip contains a router, an SDRAM and a list of processor objects, each with their respective properties. A *VirtualMachine* can also be made, which contains the same objects but can be identified as being virtual by the rest of the tools.

necessary and also to know where the connected real chip is to make use of that if needed.

4.6.2 Graphs

A graph in SpiNNTools consists of vertices and directed edges between the vertices. The vertex is considered to be a place where computation takes place, and as such, each vertex has a SpiNNaker executable binary associated with it. An edge represents some communication that will take place from a source, or pre-vertex to a target, or post-vertex. An additional concept is that of the outgoing edge partition; this is a group, or partition, of edges that all start at the same pre-vertex, as shown in Figure 4.5(b). This is useful to represent a multicast communication. Note that not all edges that have the same pre-vertex have to be in the same outgoing edge partition; there can be more than one outgoing edge partition for each source vertex. This represents different message types, which might be multicast to different sets of target vertices. Thus, each outgoing edge partition has an identifier, which can be used to identify the type of message to be multicast using that partition.

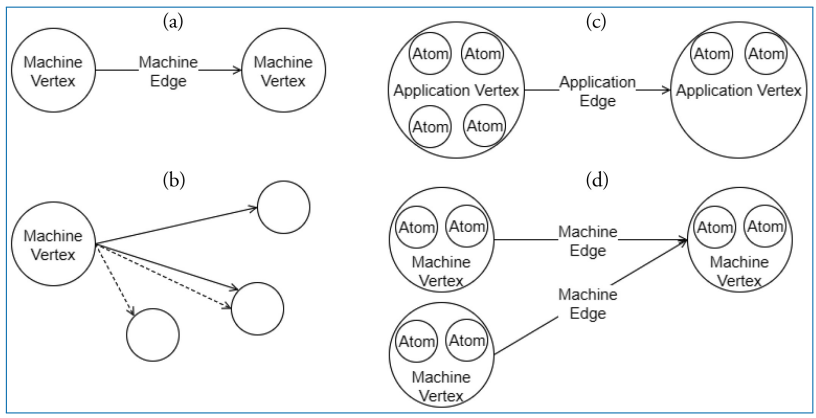

Figure 4.5. Graphs in SpiNNTools. (a) A Machine Graph made up of two Machine Vertices connected by a Machine Edge, indicating a flow of data from the first to the second. (b) A Machine Vertex sends two different types of data to two subsets of destination vertices using two different Outgoing Edge Partitions, identified by solid and dashed lines respectively. (c) An Application Graph made up of two Application Vertices, each of which contain two and four atoms, respectively, connected by an Application Edge, indicating a flow of data from the first to the second. (d) A Machine Graph created from the Application Graph in (c) by splitting the first Application Vertex into two Machine Vertices which contain two atoms each. The second Application Vertex has not been split. Machine Edges have been added so that the flow of data between the vertices in still correct.

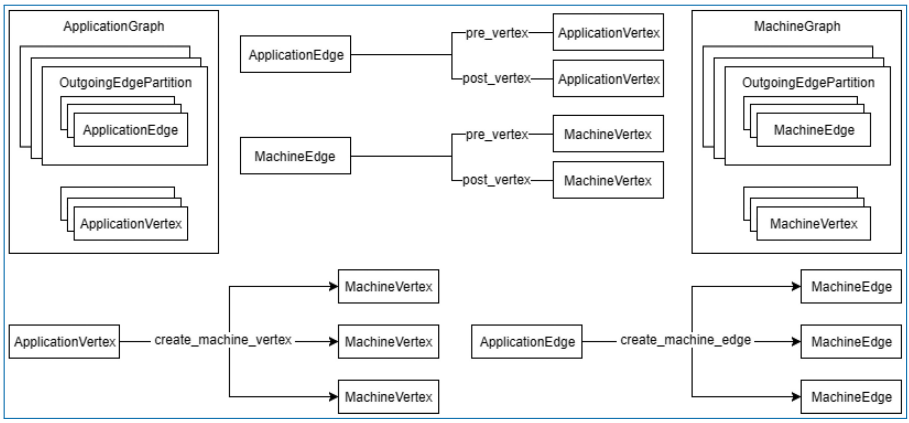

Figure 4.6. The relationship between the graph objects. An ApplicationGraph contains ApplicationVertex objects and OutgoingEdgePartition objects, which contain ApplicationEdge objects in turn. A MachineGraph similarly contains MachineVertex objects and OutgoingEdgePartition objects, which contain MachineEdge objects in turn. ApplicationEdge objects have pre- and post-vertex properties which are ApplicationVertex objects, and similarly MachineEdge objects and pre- and post-vertex properties which are MachineVertex objects. An ApplicationVertex can create a number of MachineVertex objects for a subset of the atoms contained therein and an ApplicationEdge can create a number of MachineEdge for a subset of atoms in the pre- and post-vertices.

There are two types of graph represented as Python classes in the tools (a diagram can be seen in Figure 4.6). A Machine Graph, an example of which is shown in Figure 4.5(a), is one in which each vertex (known as a Machine Vertex) is guaranteed to be able to execute on a single SpiNNaker processor. A Machine Edge therefore represents communication between cores. In contrast, an Application Graph, an example of which is shown in Figure 4.5(c), is one where each vertex (known as an Application Vertex) contains atoms, where each atom represents an atomic unit of computation into which the application can be split; it may be possible to run multiple atoms of an Application Vertex on each core. Each edge (known as an Application Edge) represents communication of data between the groups of computational units; if one or more of the atoms in an Application Vertex communicates with one or more atoms in another Application Vertex, there must be an Application Edge between those Application Vertices. It is not guaranteed that all the atoms on an Application Vertex fit on a single core, so the instruction code for Application Vertices should know how to process a subset of the atoms, and how to handle a received message and direct it to the appropriate atom or atoms. The graph classes support adding and discovering vertices, edges and outgoing edge partitions.

As the vertices represent the application code that will run on a core, they have methods to communicate their resource requirements, in terms of the amount of DTCM and SDRAM required by the application, the number of Central Processing Unit (CPU) cycles used by the instructions of the application code to maintain any time constraints, and any IP Tags or Reverse IP Tags required by the application. The Application Vertex provides a method that returns the resources required by a continuous range or slice of the atoms in the vertex; this is specific to the exact range of atoms, allowing different atoms of the vertex to require different resources. The Application Vertex additionally defines the maximum number of atoms that the application code can execute at a maximum on each core of the machine (which might be unlimited) and also the total number of atoms that the vertex represents. These allow the Application Vertex to be broken down into one or more Machine Vertices as seen in Figure 4.5(d); to this end, the Application Vertex class has a method for creating Machine Vertex objects for a continuous range of atoms. A Machine Vertex can return the resources it requires in their entirety.

The graphs additionally support the concept of a Virtual Vertex. This is a vertex that represents a device connected to a SpiNNaker machine. The Virtual Vertex indicates which chip the device is physically connected to, allowing the tool chain to work with this to include the device in the network. As with the other vertices, there is a version of the Virtual Vertex for each of the machine and application graphs.

4.7 The SpiNNTools Tool Chain

The aim of the SpiNNTools tool chain is to control the execution of a program described as a graph on the SpiNNaker machine. The software is executed in several steps as shown in Figure 4.7 and detailed below.

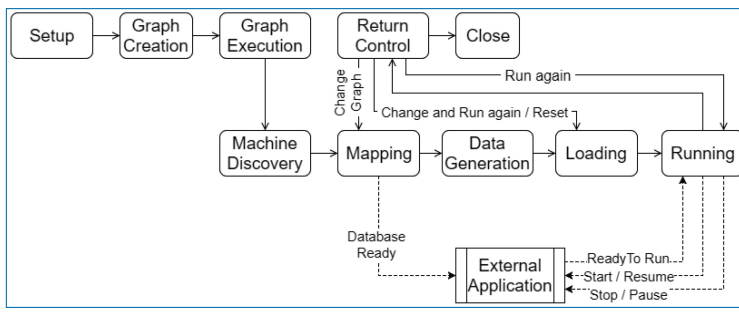

Figure 4.7. The execution work flow of SpiNNTools in use within an application. Once control has returned to the application, the flow can be resumed at different stages depending on what has changed since the last execution.

4.7.1 Setup

The first step in using SpiNNTools is to initialise them. At this point, the user can specify appropriate configuration parameters, such as the time step of the simulation, and the location where binary files can be located on the host machine. The tool chain then sets up the initially empty graphs and reads in configuration files for further options, such as the SpiNNaker machine to be used. Options are separated out in this way to allow script-level parameters which might apply no matter where the script is run (like the timestep of the simulation), from user-level parameters, which will be different per-user, but likely to be common across multiple scripts for that user (like the SpiNNaker machine to be used).

4.7.2 Graph Creation

Once the tool chain has been initialised, the user can add vertices and edges to either an application or machine graph. It is an error to add vertices or edges to both of these structures. The tool chain keeps track of the graph as it is built up. Users can extend the vertex and edge classes to add additional information relevant to their own application.

4.7.3 Graph Execution

Once the user has built their graph, they then call one of the methods provided to start execution of the graph. Methods are provided to run for a specified period of time, to run until a completion state is detected (such as all cores being in an exit state having completed some unit of work), or to run 'forever' meaning that execution can be stopped through a separate call to SpiNNTools at some indeterminate time in the future, or the execution can be left on the machine to be stopped outside of the tool chain by resetting the machine. The graph execution itself consists of several phases shown in the lower half of Figure 4.7 and detailed below.

Machine Discovery

The first phase of execution is the discovery of the machine to be executed on. If the user has configured the tool chain to run on a single physical machine, this machine is contacted, and if necessary booted. Communications with the machine then take place to discover the chips, cores and links available. This builds up a Python machine representation to be used in the rest of the tool chain.

If a machine is to be allocated, SpiNNTools must first work out how big a machine to request, by working out how many chips the user-specified graph requires. If a machine graph has been provided, this can be used directly, since the number of cores is exactly the number of vertices in the graph. The resources must still be queried, as the SDRAM requirements of the vertices might mean that

not all of the cores on each chip can be used. For example, a graph consisting of 10 machine vertices, each requiring 20 MByte of SDRAM and thus 200 MByte of SDRAM overall, will not fit on a single chip in spite of there being enough cores.

If an application graph is provided, this must first be converted into a machine graph to determine the size of the machine. This is done by executing some of the algorithms in the mapping phase (see below).

Mapping

The mapping phase takes the graph and maps it onto the discovered machine. This means that the vertices of the graph are assigned to cores on the machine, and edges of the graph are converted into communication paths through the machine. Additionally, other resources required by the vertices are mapped onto machine resources to be used within the simulation.

If the graph is an application graph, it must first be converted to a machine graph. This may have been done during the machine discovery phase as described previously. To allow this, the algorithm(s) used in this 'graph partitioning' process are kept separate from the rest of the mapping algorithms.

Once a machine graph is available, this is mapped to the machine through a series of phases. This must generate several data structures to be used later in the process. These include:

- a set of *placements* detailing which vertex is to be run on which core of the machine;
- a set of *routeing tables* detailing how communications over edges are to pass between the chips of the machine;
- a set of *routeing keys* detailing the range of keys that must be sent by each vertex to communicate over each outgoing edge partition starting at that vertex;
- a set of *IP tags* and *reverse IP tags* identifies which external communications are to take place through which Ethernet-connected chip.

Note that once machine has been discovered, mapping can be performed entirely separately from the machine using the Python machine data structures created. However, mapping could also make use of the machine itself by executing specially designed parallel mapping executables on the machine to speed up the execution. The design of these executables is left as future work.

Mapping information can be stored in a database by the system. This allows for external applications which interact with the running simulation to decode any live data received. As shown in Figure 4.7, the applications can register to be notified when the database is ready for reading and can then notify SpiNNTools when they have completed any setup and are ready for the simulation to start, and when the simulation has finished.

Data Generation

The data generation phase creates a block of data to be loaded into the SDRAM for each vertex. This can be used to pass parameters from the Python-described vertices to the application code to be executed on the machine. This can make use of the mapping information above as appropriate; for example, the *routeing keys* and *IP tags* allocated to the vertex can be passed to ensure that the correct keys and tags are used in transmission. The graph itself could also be used to determine which *routeing keys* are to be received by the vertex, and so set up appropriate actions to take upon receipt of these keys.

Some support for data generation and reading is provided by the tool chain both at the Python level, where data can be generated in 'regions', and at the C code level, where library functions are provided to access these regions. Other more basic data generation is also supported which simply writes to the SDRAM directly.

Data generation can also create a statistical description of the data to be loaded and then expand these data through the execution of a binary on the machine. This allows less data to be created at this point potentially speeding up the data generation and loading processes, and also allows the expansion itself to occur in parallel on the machine.

Loading

The loading phase takes all the mapping information and data generated, along with the application binaries associated with each machine vertex, and prepares the physical machine for execution. This includes loading the *routeing tables* generated on to each chip of the machine, loading the application data into the SDRAM of the machine, loading the *IP tags* and *reverse IP tags* into the Ethernet chips, and loading the application code to be executed.

Running

The running phase starts off the actual execution of the simulation and, if necessary, monitors the execution until complete. Before execution, the tool chain wait for the completion of the setup of any external applications that have registered to read the mapping database. These tools are then notified that the application is about to start, and when it is finished.

Once a run is complete, application recorded data and provenance data are extracted from the machine. The provenance data include:

- router statistics, including dropped multicast packets;
- core-level execution statistics, including information on whether the core has kept up with timing requirements;

- custom core-level statistics, these depend on the application, but might include such things as the number of spikes sent in a neural simulation or the number of times a certain condition has occurred.

The log files from each core can also optionally be extracted. During provenance extraction, each vertex can analyse the data and report any anomalies. If the log files have been extracted, these can also be analysed and any 'error' or 'warning' lines can then be printed.

If a run is detected to have failed in some way, the tool chain will attempt to extract information about this failure. A failure includes one of the cores going into an error state, or if the tool chain have been run for a specific duration, if the cores are not in a completion state after this time has passed. Log files will be automatically extracted here and analysed as previously discussed. Any cores that are still alive will also be asked to stop and extract any provenance data so that this can also be analysed in an attempt to diagnose the cause of the error.

The run may be split into several sub-runs to allow for the limited SDRAM on the machine, as shown in Figure 4.8. After each run cycle, any recorded data are extracted from the SDRAM and stored on the host machine, after which the recording space is flushed, and the run cycle restarted. This requires additional support within the binary of the vertex, to allow a message to be sent to the core to increase the run duration, and to reset the recording state. This support is provided in the form of C code library functions, with callbacks to allow the user to perform

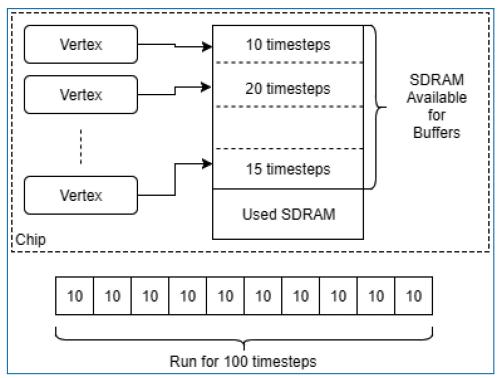

Figure 4.8. Running vertices with recorded data. The SDRAM remaining on each chip after it has been allocated for other things is divided up between the vertices on that chip. Each vertex is then checked for the number of time steps it can be run for before filling up the SDRAM. The minimum number of time steps is taken over all chips and the total run time is split into smaller chunks, between which the recorded data are extracted and the buffer is cleared.

additional tasks before resuming execution at each phase. Additionally, the tool chain can be set up to extract and clear the core logs after each run cycle to ensure that the logs do not overflow.

The length of each run cycle can be determined automatically by SpiNNTools. This is done by working out the SDRAM available on each chip after data generation has taken place. This free space is then divided between the vertices on the chip depending on how much space they require to record per time step of simulation. To ensure that there is some space for recording, the user can specify the minimum number of time steps to be recorded and space for this is allocated statically during the mapping phase (noting that if this space cannot be allocated, this phase will fail with an error).

At the end of each run phase, external applications are notified that the simulation has been paused and are then notified again when the simulation resumes. This allows them to keep in synchronisation with the rest of the application.

4.7.4 Return of Control/Extraction of Results

Once the run cycles have completed, the tool chain returns control to the executing script. At this point, the user can interact with the graph again. This includes the ability to extract any recorded data (see later) or make changes to the graph and/or the parameters before resuming the simulation. The effect of any changes is detailed below.

4.7.5 Resuming/Running Again

The user can choose to resume the execution of the simulation or to reset the simulation and start it again. At this point, the tool chain must decide which of the aforementioned steps need to be run again. If no changes have been made to the graph or the parameters, this can simply be considered an extension of the aforementioned ability of the SpiNNTools to run the code in phases. The minimum time calculated previously is respected again here and the tool chain will then run in cycles of this unit of time. Note that this means that if the first run-time is shorter than that required to fill the remaining SDRAM space (and thus only one run cycle was required previously), this time is taken as the minimum. This is because the buffers will have already been initialised to record for this amount of time. An extension to this work then is to allow the buffers to be sized to use up all of the remaining SDRAM regardless of the run time and then allow runs in units of less than or equal to the time that uses all of this space.

If the parameters of any of the vertices or edges have been changed, the vertex can be set up to allow the reloading of these changes. It is expected that this can be supported where the change will not increase the size of the data, and so can

overwrite the existing data, such as a change in neuron state update parameters in a neural network. Any increase in the size of the data, such as an increase in the number of synapses in a neural network, would likely require a remapping of the graph on to the machine as the SDRAM is likely to be packed in such a way as to not allow the expansion of the data for a single core; it is left to the vertex to make this decision however.

Any change to the graph, such as the addition of a vertex or edge, is likely to require that the mapping phase take place again. This may even result in a new machine being required should the size of the graph increase to this degree. This will mean that all the other phases will also have to be executed again.

4.7.6 Closing

Once the user has finished simulating and extracted any data, they can tell the tool chain that they are finished with the machine by closing it. At this point, the tool chain resets and releases any machines that have been reserved, and so recorded data will no longer be available. If the tool chain was told to run the network for an indeterminate length, this would also result in the extraction and evaluation of any provenance data at this stage.

4.7.7 Algorithms and Execution

To run each of the above phases, SpiNNTools executes a series of algorithms. The algorithms consume various inputs that are made available by the tool chain and by other algorithms, and produce various outputs. These inputs and outputs are not constrained in any other way; thus, algorithms are not constrained to produce only one output. This could be useful in, for example, mapping, where an algorithm could be made to produce both placements and routeing tables which have been optimised together. This is in contrast to restricting the algorithms to specific tasks, where the output might then be less optimal, such as having a specific algorithm for generating placement and another for generating routeing tables.

To support this form of execution, SpiNNTools implements a workflow execution system, shown in Figure 4.9. This examines the algorithms to be run in terms of the inputs required and outputs generated to compute an execution order for the algorithms. Input and output 'tokens' are also supported; these indicate implicit inputs and outputs; for example, a token might be used to represent that data have been loaded on to the machine, and thus, an algorithm can generate this as an output, and another can require that this has been completed before execution.

The algorithms themselves are not discussed here in detail other than those mentioned above. A more detailed discussion of the mapping algorithms is discussed

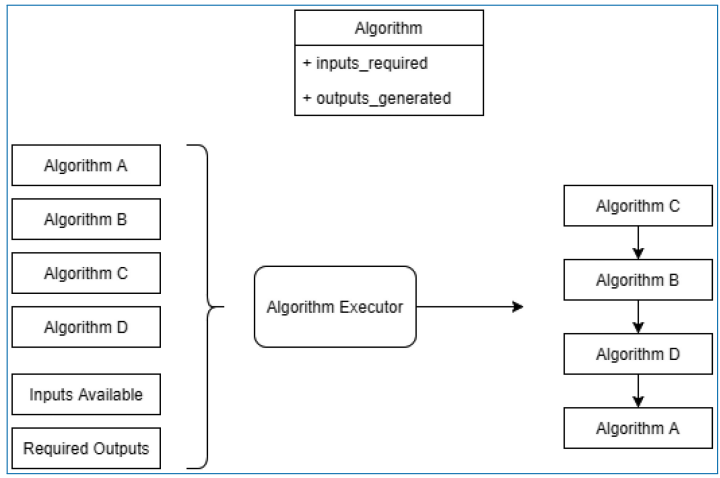

Figure 4.9. Algorithms being run by the algorithm execution engine. The executor is provided with a list of algorithms to run, a set of input items and a set of output items to produce. It then produces a workflow for the algorithms accounting for their inputs required and outputs produced.

by Heathcote [94]. The tool chain also includes algorithms for routeing table compression, which are discussed by Mundy *et al.* [174]. Many of the other algorithms are currently simplistic in nature; these can be replaced in the future should other algorithms be found to perform more efficiently and/or effectively.

4.7.8 Data Recording and Extraction

As mentioned previously, the tool chain supports the recording of data in such a way as to cope with the limited nature of the SDRAM on the machine. A 'buffer manager' is provided, which is used to keep track of and store the buffers of data as they are extracted from the machine. This can additionally support the live extraction of buffers whilst the simulation is running, as shown in Figure 4.10 (Top); cores configured with the provided library can contact the host machine when the recording space is getting full and the tool chain can then attempt to extract the data. In general, the bandwidth of the Ethernet of the machine is not fast enough for this to be effective, and data tend to be lost.

The SCAMP software supports the reading of SDRAM through SDP messages. This works through a request and response system, where each SDP message can request the reading of up to 256 bytes of data. Additionally, to transmit the SDP message to chips which are not connected to the Ethernet, this message must be broken down into SpiNNaker network messages and then reconstructed on receipt; an overview of how this process works is shown in Figure 4.10 (Middle). This

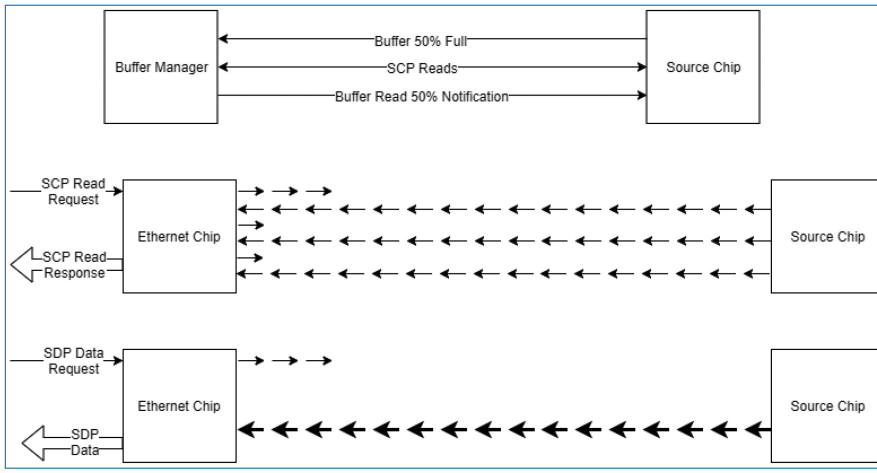

Figure 4.10. Data buffering and extraction. *Top:* The buffer manager is used to read back recorded data during execution; when the buffer contains some data, the buffer manager is notified and attempts to read the data, notifying the data source once this has been done to allow the space to be reused. *Middle:* Data reading done using SCAMP; each read of up to 256 bytes is further broken down into a number of request and read cycles on the machine itself, where the packets used contain only 24 bits of data each. *Bottom:* Data reading done using multicast messages; the initial request is all that is required, after which the data are streamed using packets containing 64 bits of data. The machine is set up so that these packets are guaranteed to arrive, so no confirmation is required.

results in speeds of around 8 Mb/s when reading from the Ethernet chip and around 2 Mb/s when reading from other chips.

To speed up the extraction of data, the tool chain includes the ability to circumvent this process, an overview of which is shown in Figure 4.10 (Bottom). To facilitate this, firstly the machine is configured so that packets can be sent with a guarantee that none of them are ever dropped; this can be done in this scenario because exactly one path through the machine will be used by each read, so deadlocks cannot occur. Next, one of the cores on each chip is loaded with an application that can read from SDRAM and stream multicast messages to another application loaded onto a core on the Ethernet chip, which then forms these into SDP messages to be streamed to the host along with a sequence number in each SDP packet. The host then gathers the SDP packets and notes which sequences are missing. The missing sequences are then requested again from the machine; this is repeated until all sequences have been received. This has numerous advantages over the SDP request-and-response mechanism: the SDP is only formed at the Ethernet chip, and thus, the headers do not get transmitted across the SpiNNaker fabric; and the host only sends in a single request for data and then a single request for each group

of missing sequences and thus does not have to wait for each chunk of 256 bytes between sending requests. This results in speeds of up to 40 Mb/s when reading from any chip on the machine; there is no penalty for reading from a non-Ethernet chip.

Once this protocol was implemented, we discovered that the Python code had trouble keeping up with the speed at which the data were received from the machine. We therefore implemented a version of the data reception in C++ and Java that could interface with the Python code; the Java version is the version used in production following comparative testing and assessment of the integration quality. This then allows the use of the Ethernet connection on multiple boards simultaneously, allowing the data extraction speed to scale with the number of boards required for the simulation, up to the bandwidth of the network connected to the host machine.

4.7.9 Live Interaction

We have previously mentioned that external applications can interact with a live simulation, making use of the mapping database. Additional support for this interaction is provided by the tool chain. This support is split into live data output and live data input.

Live data output support is performed by a vertex called the 'Live Packet Gatherer', which will package up any multicast packets it receives and send them as UDP packets using the EIEIO protocol [205]. It is configured by adding edges to the graph from vertices that wish to output their data in this way. This has the advantage of being able to tap into the existing multicast streams that are already being used to communicate within the machine; this same data can be sent out of the machine by the simple addition of an edge to the graph, as shown in Figure 4.11.

Live data input support is provided via a vertex called the 'Reverse IP Tag Multicast Source', which will unpack and send multicast packets using the same EIEIO protocol. As with the Live Packet Gatherer, this vertex can then be configured by simply adding edges from it to the vertices which are to receive the messages.

External applications that would like to make use of this support can read the mapping database to determine the multicast keys to be received in the case of live output or to be sent in the case of live input. Support for this interaction is provided in SpiNNTools in both Python code and host-based C++ code.

4.7.10 Dropped Packet Re-Injection

As mentioned in Section 4.2, when a packet is dropped, an interrupt is raised allowing a core to detect and capture the dropped packet. The tool chain includes software that runs on the SpiNNaker machine to detect this interrupt and then

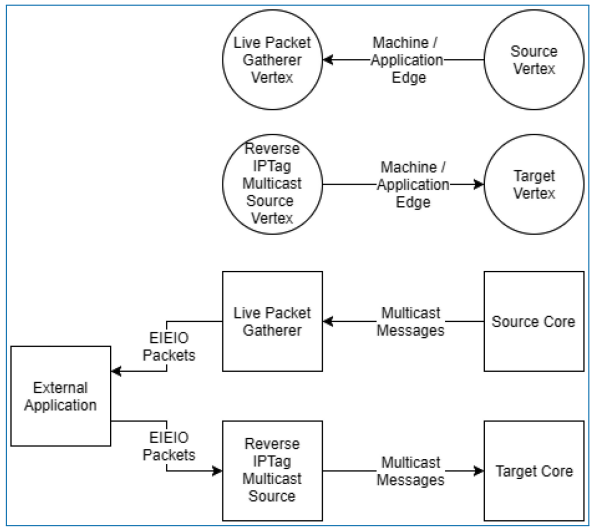

Figure 4.11. Live interaction with vertices. *Top:* To indicate that live output is required, an edge is added from the vertex which is the source of the data to the Live Packet Gatherer vertex in the graph. To indicate that the live input is required, an edge is added from the Reverse IP Tag Multicast Source vertex to the target of the data in the graph. *Bottom:* The effect of adding the edges to the graph is that multicast messages will be sent from the core (or cores) of the source vertex to the core running the Live Packet Gatherer, which will then wrap the messages in EIEIO packets and forward them to a listening external application; and EIEIO packets received from an external application will be decoded by the Reverse IP Tag Multicast Source core and dispatched as multicast messages to the target core (or cores).

capture the packets that have been dropped. These are stored until a time at which the router is no longer blocked and so can safely send the packet onwards. This helps in those applications where the reliable transmission of packets is critical to their operation.

There is only one register within the SpiNNaker hardware to hold a dropped packet. If a second packet is dropped, this packet will be completely unrecoverable; an additional flag is set in this scenario so the re-injection core can detect this and count such occurrences. This count is reported to the user at the end of the execution so that they know that something may not be correct in their simulation results.

4.7.11 Network Traffic Visualisation

A real-time traffic visualiser for a single 48-node SpiNN-5 board was developed to explore the control and monitoring of the SpiNNaker system in real time [144].

The visualiser shows the system traffic status by gathering and displaying data from the monitoring and profiling counters on the SpiNNaker chips in the system. The visualiser can also send commands to the monitor processor via the Ethernet connection to control and interact with the system.

4.7.12 Performance and Power Measurements

The tool chain includes support for profiling any executed code and for making an estimate of the power usage of a simulation. Profiling support is provided through both C and Python libraries, where the former is used to instrument code with 'entry' and 'exit' markers for code to be timed, and the latter is used to extract the recorded timing data and calculate various statistics on the run.

To provide a reasonably accurate power estimation, the tool chain includes support for sampling the *System Controller* to determine whether each core is busy or idle (waiting for an event to occur), and we include a uniformly-distributed random delay to the sampling to avoid the worst effects of sample aliasing. As this is run on the machine, it can achieve a higher sampling rate than a commercial power-measurement tools. We then use the proportion of time spent idling, together with the number of SpiNNaker messages sent, to compute the estimate for how much power was actually used, scaling by the previously measured long-term average power consumption per core and per message. This has been tested against a commercial power measurement device on a 24-board system and appears to provide results close to the real numbers.

4.8 Non-Neural Use Case: Conway's Game of Life

Conway's Game of Life [71] consists of a collection of cells which are either alive or dead based on the state of their neighbouring cells. A diagram of an example Machine Graph of this problem is shown in Figure 4.12. The vertices of the graph of this application are each a cell in the game; given the state of the surrounding cells, this cell can compute whether it is dead or alive in each step and then send that to its neighbours. It similarly receives the state of the neighbours as they are transmitted and again uses this to update its own state. The edges of the graph are thus between adjacent cells in a grid, where each vertex is connected bidirectionally to its eight surrounding neighbours. The game proceeds in synchronous phases, where the state of cells in a given phase are all considered at the same time.

Graphs of this form are highly scalable on the SpiNNaker system, since the computation to be performed at each node is fixed, and the communication forms a regular pattern which does not increase as the size of the board grows. Thus once

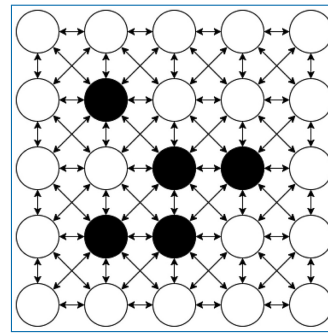

Figure 4.12. Conway's Game of Life on a 5 × 5 grid as a Machine Graph. Every Machine Vertex is connected to each of it's 8 neighbours bi-directionally; this requires two Machine Edges for each bi-directional connection. The initial state of each Vertex is either alive (black) or dead (white).

working, it is likely that any size of game can be built, up to the size of the available machine. This type of graph would also likely be suited to finite element analysis [17] problems, provided that the data to be transmitted can be broken down into SpiNNaker packets. This problem thus works well as an archetype.

It will be assumed that we have built the application code which will update the cell based on the state of the surrounding cells. This will update the state once per time step of the simulation based on the received state from the surrounding cells and then send its own new state out using the given key. It can also record its state at each time step in the simulation. The set-up of this application is as follows:

- A Conway vertex is created which extends the machine vertex class.
- A number of Conway vertices are added to the graph to make up the board. These are stored in such a way that finding an adjacent vertex in the grid is easy.
- A machine edge is added between each pair of adjacent vertices, in each direction.
- Each machine vertex generates data for the vertex, which includes the key to be sent by that vertex and the number of time steps to run for.
- Each machine vertex can tell the tool chain how many time steps it can run for given an amount of SDRAM available for recording.
- Each machine vertex contains code to read the state that is recorded at each time step using the Buffer Manager.

Once the graph is built, the script starts the execution of the graph. During this execution, the tool chain will obtain a machine description and use this with the

machine graph to work out a placement of each of the vertices and a routeing of the edges between these placements, along with an allocated key for each of the vertices. The software tools will then ask each vertex how many time steps it can record for based on the available SDRAM after placement is complete, and the resources used on each chip can therefore be determined. Each vertex will then be asked to generate its data based on the mapping and timing information. SpiNNTools will then load the generated data onto the machine along with the routeing tables and application code and start the execution of the cores. It will wait an appropriate amount of time for the cores to stop and then check their status. Assuming this is successful, control will return to the script. This can then request the recorded states from each of the vertices and display these data in an appropriate way.

A future version could have a Conway vertex that can have multiple cells within each machine vertex, which would then allow for an application vertex of cells. This would have a single large Application Vertex which would represent the whole game board and an Application Edge for each of the 8 directions of connectivity, each in its own Outgoing Edge Partition to indicate that different keys are required for each of the directions. This would require that the vertex would have to cope with the reception of multiple neighbour states, which would make the application code itself more complex; for example, it would have to cope with multiple incoming keys from each direction, each of which would target a different cell within the grid.

Another possible extension to this application is to extract the state during execution and display this as the application progresses. This would require the addition of the Live Packet Gatherer vertex (described above) to the graph and an edge from each of the Conway vertices to this vertex. The script would then indicate, before executing the graph, that there is an external application that would like to receive the data. This application will receive a message when the mapping database has been written, at which point, it can set up a mapping between multicast keys received and positions in the game board, responding when it has completed its own setup. The tool chain will then notify this application that the simulation is starting, and the application will then receive the same state messages as the vertices receive, which it can use to update the display of the game board.

4.9 sPyNNaker – Software for Modelling Spiking Neural Networks

The SpiNNaker machine is primarily designed to simulate spiking neural networks [65]. As an example, we consider the simulation of a cortical column found within mammalian brains, that is, a model of the neurons within a structure underneath

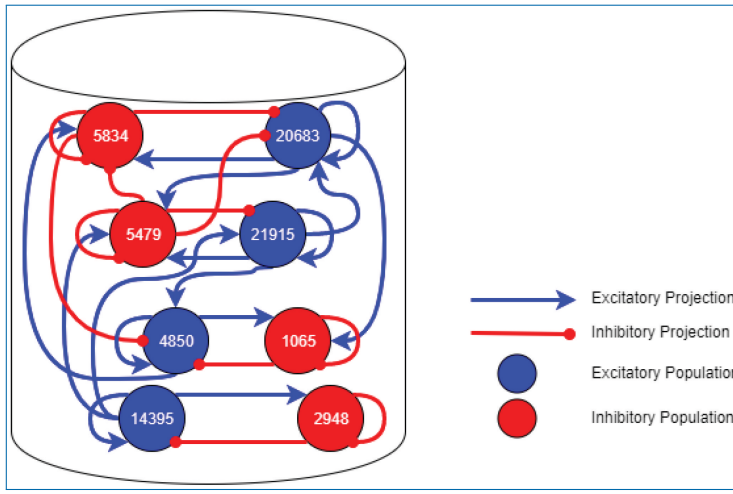

Figure 4.13. A neural network topology of a 1 mm^2 area of cortical microcircuit found within the mammalian brain. Each population of neurons is shown as a circle containing a number, where the number indicates the number of neurons in that population.

a 1 mm^2 area of the surface of the generic early sensory cortex [201]. Figure 4.13 shows the groups of neurons (Populations) in this network and the connectivity between them (Projections). In a spiking neural network, the vertices are groups of point neurons (as a single core can simulate more than one neuron); the computation required is the update of the neuron state in response to spikes received from connected neurons. The edges are then groups of synapses between the neurons, over which spikes are transmitted. These are potentially unidirectional and are likely to be more heterogeneous in nature than the regular grid pattern seen in Conway's Game of Life.

The problem of SNNs is clearly well suited to the architecture, as this is what it was designed for, but the heterogeneity of the network, and the fact that multiple neurons are computed on each core means that some networks will be more suited to the platform than others; in particular, neural networks often form 'small world' networking topologies, where most of the connections are relatively local, but there are a few long-distance connections. The computation required to simulate each neuron at each time step in the simulation is generally fixed. The remaining time is then dedicated to processing the spikes received, the number of which depends on the how many neurons are sending spikes to the core and the activity of those connected neurons. This is not known in advance in general, so some flexibility in the system with respect to the amount of computation available at each node is necessary to allow the application to work in different circumstances. Once this

is known for a given network, the system could potentially be reconfigured with additional cores, allowing that network to be simulated in less time overall.

4.9.1 PyNN

PyNN is a Python interface to define SNN simulations for a range of simulator back-ends [44]. It allows users to specify an SNN simulation via a Python script once and have it executed on any or all of the supported back-ends including NEST [76], NEURON [33] and Brian [82]. This encourages standardisation of simulators and reproducibility of results, and increases productivity of neural network modellers through code sharing and reuse, by providing a foundation for simulator-agnostic post-processing, visualisation and data-management tools.

PyNN has continued development as part of the European Flagship Human Brain Project (HBP) [4], and has hence been adopted as a modelling language by a number of partners including SpiNNaker. It provides a structured interface for the definition of neurons, synapses and input sources, giving users the flexibility to build a range of network topologies. Models typically consist of single-compartment point neurons, grouped together in *populations*. These populations are then linked with *projections*, representing the synaptic connections between the axons of neurons in a source population, and the dendrites of neurons in a target population. Once defined, a number of simulation controls are used to execute the model for a given time period, with the option to update parameters and initialise state variables between runs. On simulation completion, data can be extracted for post-processing and future reference. Neuron variables such as spike trains, total synaptic conductances and neuron membrane potential are accessible from population objects, while synaptic weights and delays are extracted from projections. These data can be subsequently saved or visualised using the built-in plotting functionality.

Example PyNN commands for the generation of populations and projections are detailed in Listing 4.1. Here the sPyNNaker version of the simulator is imported as *sim* and subsequently used to construct and execute a simulation. A population of 250 Poisson source neurons is created with label 'poisson_source' and provides 50 Hz input to the network for 5 s. A second population of 500 integrate and fire neurons is then created and labelled as 'excitatory_pop'. Excitatory connections are made between 'poisson_source' and 'excitatory_pop' with a 20% probability of connection, each with a weight of 0.06 nA and delays specified via a probability distribution. Data recording is then enabled for 'excitatory_pop', and the simulation is executed for 5 s. Finally, the 'excitatory_pop' spike history data are extracted from the simulator.

```
import pyNN.spiNNaker as sim
# Spike input
poisson_spike_source = sim.Population(250, sim.SpikeSourcePoisson(
    rate=50, duration=5000), label='poisson_source')
# Neuronal populations
pop_exc = sim.Population(500, sim.IF_curr_exp(**cell_params_exc),
                         label='excitatory_pop')
# Poisson source projections
poisson_projection_exc = sim.Projection(poisson_spike_source, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.06, delay=delay_distribution),
    receptor_type='excitatory')
# Specify output recording
pop_exc.record('all')
# Run simulation
sim.run(simtime=5000)
# Extract results data
exc_data = pop_exc.get_data('spikes')
```

Listing 4.1. Example PyNN commands (a complete script is detailed in Listing 4.2).

The job of a PyNN simulator is therefore to provide a back-end-specific implementation of the PyNN language, enabling execution of simulations defined in model scripts such as Listing 4.2.

4.9.2 sPyNNaker Implementation

The sPyNNaker Application Programming Interface (API) is comprised of two software stacks as shown in Figure 4.14: one running on host predominantly written in Python, the other running on the SpiNNaker machine written in C.

4.9.3 Preprocessing

At the top of the left-hand side stack in Figure 4.14, users create a PyNN script defining an SNN. The SpiNNaker back-end is specified, which translates the SNN into a form suitable for execution on a SpiNNaker machine. This process includes mapping of the SNN into an application graph, partitioning into a machine graph, generation of the required routeing information and loading of data and applications to a SpiNNaker machine. Once loading is complete, all core applications are instructed to begin execution and run for a predefined period. On simulation completion, requested output data are extracted from the machine and made accessible through the PyNN API.

A sample SNN is developed as a vehicle by which to describe the stages of preprocessing. A random balanced network is defined according to the PyNN script detailed in Listing 4.2, with the resulting network topology shown in Figure 4.15(a). The network consists of 500 excitatory and 125 inhibitory neurons, which make excitatory and inhibitory projections to one another, respectively. Each population additionally makes recurrent connections to itself with the same effect. Excitatory Poisson-distributed input is included to represent background

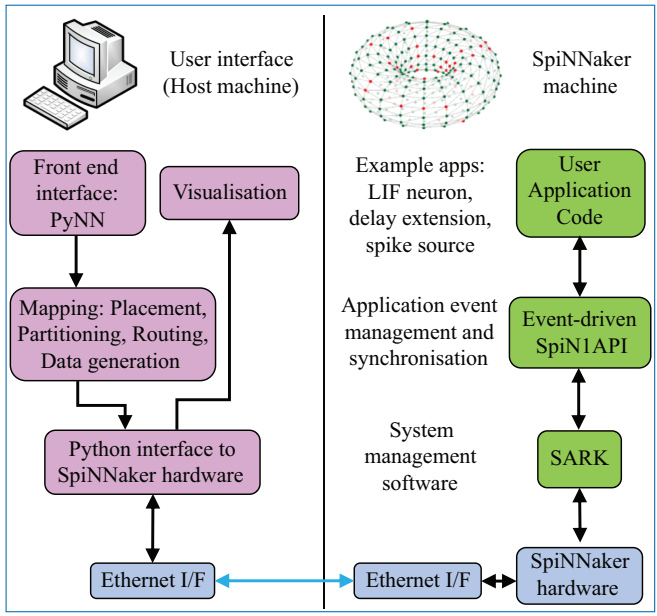

Figure 4.14. SpiNNaker software stacks. From top left anti-clockwise to top right: users create SNN models on host via the PyNN interface; the sPyNNaker Python software stack then translates the SNN model into a form suitable for a SpiNNaker machine and loads the appropriate data to SpiNNaker memory via Ethernet; sPyNNaker applications, built on the SARK system management and SpiN1API event-driven processing libraries, use the loaded data to perform real-time simulation of neurons and synapses.

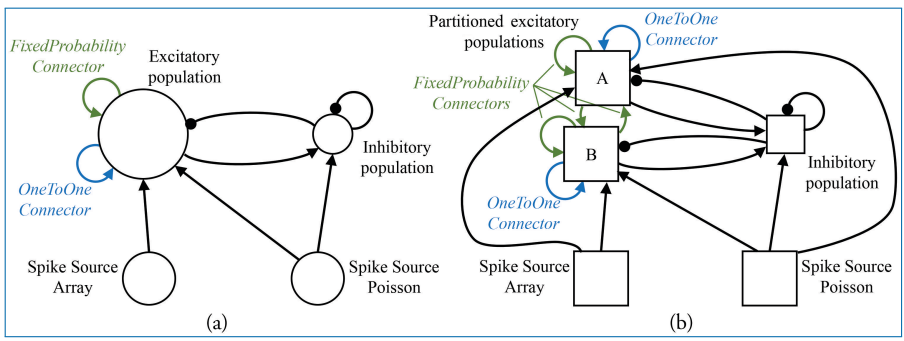

Figure 4.15. Network partitioning to fit machine resources. (a) Application graph generated from interpretation of PyNN script: circles represent PyNN populations, and arrows represent PyNN projections. (b) Machine graph partitioned into vertices and edges to suit machine resources: squares represent populations (or partitioned sub-populations) of neurons which fit on a single SpiNNaker core – hence, the model described by the machine graph in (b) requires 5 SpiNNaker cores for execution.

activity, while predefined spike patterns are injected via a spike source array. The neuronal populations consist of current-based Leaky Integrate and Fire (LIF) neurons, with the membrane potential of each neuron in the excitatory population initialised via a uniform distribution bounded by the threshold and resting potentials. The sPyNNaker API first interprets the PyNN defined network to construct an application graph: a vertices and edges view of the neural network, where each edge corresponds to a projection carrying synapses, and each vertex corresponds to a population of neurons. This application graph is then partitioned into a machine graph, by subdividing application vertices and edges based on available hardware resources and requirement constraints, ultimately ensuring each resulting machine vertex can be executed on a single SpiNNaker core. From hereon, the term *vertex* will refer to a machine vertex and is synonymous with the term sub-population, representing a group of neurons which can be simulated on a single core. An example of this partitioning is shown in Figure 4.15, where due to its size 'excitatory population' is split into two sub-partitions (A and B). Figure 4.15 also shows how additional machine edges are created to preserve network topology between partitions A, B, and the other populations, and how different PyNN connectors are treated differently during this process. For example, a PyNN *OneToOneConnector* connects each neuron in a population to itself. This results in both partitions A and B having a machine edge representing their own connections, but with no edge required to map the connector from one sub-population to the other. Conversely, the PyNN *FixedProbabilityConnector* links neurons in the source and target populations based on connection probability and hence requires machine edges to carry all possible synaptic connections (e.g. both between vertices A and B, and to themselves).

Once partitioned, the machine graph is placed onto a virtual representation of a SpiNNaker machine to facilitate allocation of chip-based resources such as cores and memory. Known failed cores, chips and board links which compromise the performance of a SpiNNaker machine are removed from this virtual representation, and the machine graph is placed accordingly. Chip-specific routeing tables are then generated facilitating transmission of spikes according to the machine edges representing the PyNN-defined projections. These tables are subsequently compressed and loaded into router memory (as described in the previous chapter). The Python software stack from Figure 4.14 then generates the core-specific neuron and synapse data structures and loads them onto the SpiNNaker machine using the SpiNNTools software. Core-specific neuron data are loaded to the appropriate DTCM, while the associated synapse data are loaded into core-specific regions of SDRAM on the same chip, ready for use according to Section 4.9.4. Finally, programs for execution on application cores are loaded to ITCM, with each core executing an initialisation function to load appropriate data structures (from SDRAM) and prepare the core

before switching to a *ready* state. Once all simulation cores are *ready*, the signal to begin simulation is given to all cores from host, and the SNN will execute according to the processes defined in Section 4.9.4.

4.9.4 SpiNNaker Runtime Execution

sPyNNaker applications execute SNNs via a hybrid simulation approach, using time-driven neuron updates and event-driven synapse updates, similar to that discussed by Morrison *et al.* [172]. This neuron update scheme provides a flexible framework in which to embed a range of neuron models and is of comparable efficiency to event-based approaches when considering biologically representative spike rates. Synapse events are handled efficiently, with no intermediate information required to update synaptic state between pre-synaptic neuron spikes, which are relatively infrequent on the order of 1 Hz in biological networks. Cores executing sPyNNaker applications hold neuron state variables in local DTCM, allowing efficient access to the required data structures for the periodic time-driven neuron update. Spike transmission between cores is via the AER model [158], with neuronal action potentials communicated as multicast packets, with their key containing only the source neuron ID (in the remainder of this work, the terms: action potential, spike and packet are synonymous). Each packet can be delivered to multiple locations simultaneously via the SpiNNaker routeing fabric, replicating the one-to-many connectivity of an axon. Processing of the packet is performed by the core simulating the post-synaptic neuron, which contains functions to evaluate the spike-based synaptic contribution using only the packet key. Due to the potentially large fan-in to a neuron, memory constraints prevent storage of synaptic data in DTCM. Therefore, the source neuron ID is used to locate the associated synaptic data stored in the relatively large but slower SDRAM memory and copy it locally on spike arrival to facilitate evaluation of the contribution to the synaptic state.

This section focuses on the deployment of this simulation approach within a single core modelling a sub-population of neurons, such as 'Excitatory A' in Figure 4.15(b).

Using the Low-Level Libraries

sPyNNaker applications are compiled against the aforementioned SpiNNaker Application Runtime Kernel (SARK) [251] and the event-driven library SpiN1API [223, 234], as shown in Figure 4.16(a).

In sPyNNaker applications modelling systems of neurons and synapses, callbacks are registered against hardware events: *timer*, *packet received* and *DMA complete*; and a software-triggered *user* event, as shown in Table 4.1. The associated callbacks

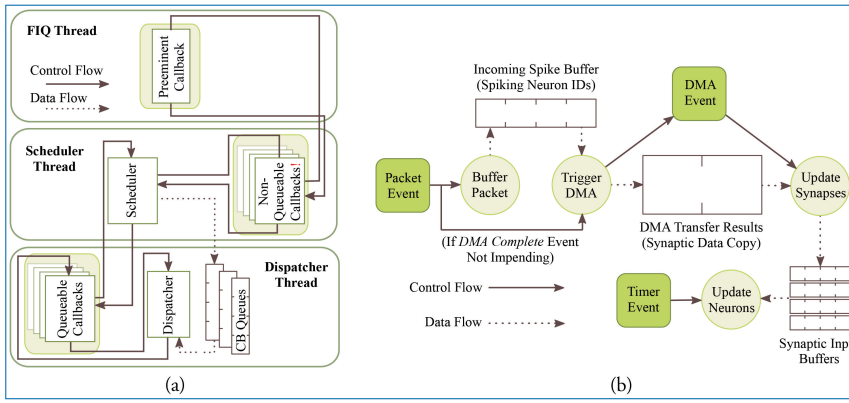

Figure 4.16. SpiNNaker realtime OS: (a) SpiN1API multi-threaded event-based operating system: scheduler thread to queue callbacks; dispatcher thread to execute callbacks; and FIQ thread to service interrupts from high-priority (–1) events. (b) Events and associated callbacks for updating neuron state variables and processing incoming packets representing spikes into synaptic input. Figures reproduced with permission from [222, 223].

Table 4.1. Hardware (and single software) events, along with their registered callback and associated priority level.

Event	Callback	Priority	Pre-empts priority
Packet received	_multicast_packet_received_callback	−1	0, 1, 2
DMA complete	_dma_complete_callback	0	1, 2
Timer	timer_callback	2	–
User (Software)	user_callback	0	1, 2

facilitate the periodic updating of neuron state and the event-based processing of synapses when packets representing spikes arrive at a core. These events (squares) and their callbacks (circles) are shown schematically in Figure 4.16(b). The function timer_callback evolves the state of neurons in time and is called periodically against *timer* events throughout a simulation. A *packet received* event triggers a _multicast_packet_received_callback, which reads the packet to extract and transfer the source neuron ID to a spike queue. If no spike processing is currently being performed, the software-triggered *user* event is issued and, in turn, executes a user_callback that reads the next ID from the spike queue, locates the associated synaptic information stored in SDRAM and initiates a DMA to copy it into DTCM for subsequent processing. Finally, the _dma_complete_callback is executed on a *DMA complete* event and initiates processing of the synaptic contribution(s) to the post-synaptic neuron(s). If on completion of this processing there are items remaining in the input spike queue, this callback initiates processing of

the next spike: meaning this collection of callbacks can be thought of as a spike processing pipeline.

Time-Driven Neuron Update

A sPyNNaker simulation typically contains multiple cores, each simulating a different population of neurons (see Figure 4.15(b)). Each core updates the states of its neurons in time via an explicit update scheme with fixed simulation timestep (Δt). When a neuron is deemed to have fired, packets are delivered to all cores that neuron projects to and processed in real time by the post-synaptic core to evaluate the resulting synaptic contribution. Therefore, while all cores operate asynchronously, it is desirable to advance neurons on all cores approximately in parallel to march forward a simulation coherently. All cores in a simulation therefore start synchronised and register *timer* events with common frequency, with the period between events defined by a fixed number of clock cycles, as shown in Figure 4.17. All cores will therefore initiate a *timer* event and execute a timer_callback to advance the state of their neurons approximately in parallel, although the system is asynchronous as there is no hardware or software mechanism to synchronise cores. Individual update times may vary due to any additional spike processing (see Section 4.9.4); however, cores that have additional spikes to process between one pair of timer events can catch up during subsequent periods of lower activity. Relative drift between boards

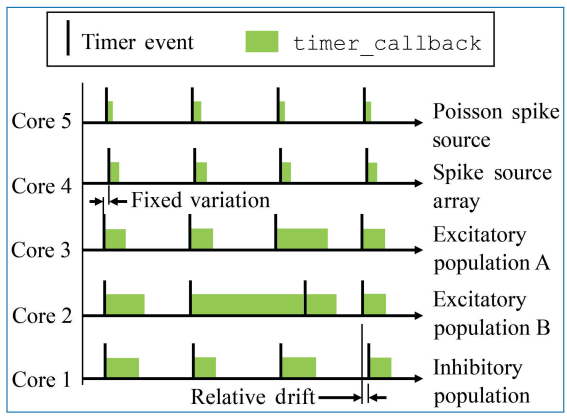

Figure 4.17. Time-driven updates by neuron cores simulating the network in Figure 4.15(b): periodic *timer* events trigger callbacks advancing neuron states by Δt. Cores can be out of phase due to communication of the start signal, and relative drift can occur due to manufacturing variability between boards. Note that state update times vary with the level of additional spike processing within a simulation timestep, however cores which experience high levels of spike activity delaying the subsequent time_callback can catch up during subsequent periods of lower spike activity (as shown by Core 2).

is possible due to slight variations in clock speed (from clock crystal manufacturing variability); however, this effect is small relative to simulation times [235]. Small variations placing core updates slightly out of phase can also occur due to the way the 'start' signal is communicated, particularly on larger machines; however, again this effect is negligible. A consequence of this update scheme is that generated spikes are constrained to the time grid (multiples of the simulation timestep Δt). It also enforces a finite minimum simulation spike transit time between neurons of Δt, as input cannot be guaranteed to arrive in the current timestep before a neuron has been updated. From the hardware perspective, the maximum packet transit time for the million core machine is $\leq 25\ \mu$s (assuming 200 ns per router [235], and a maximum path length of 128).

A design goal of the SpiNNaker platform is to achieve real-time simulation of SNNs, where 'real time' is defined as when the time taken to simulate a network matches the amount of time the network has modelled. Therefore, an SNN with a simulation timestep of $\Delta t = 1$ ms requires the period of *timer* events to be set at 200,000 clock cycles (where at 200 MHz each clock cycle has a period of 5 ns – see Section 2.2). This causes 1 ms of simulation to be executed in 1 ms, meaning the solution will keep up with wall-clock time, enabling advantageous performance, and interaction with systems operating on the same clock (such as robots, humans and animals). In practice, real-time execution is not always possible, and therefore, users are free to reduce the value of Δt in special cases and also adjust the number of clock cycles between *timer* events. For example, if a neuron model requires $\Delta t = 0.1$ ms for accuracy, it is a common practice to let the period between *timer* events remain at 200,000 clock cycles, to ensure there is sufficient processing time to update the neurons and process incoming spikes [217]. This enforces a slowdown factor of 10 relative to real time.

From the perspective of an individual core, each neuron is initialised with user-defined parameters at time t_0 (supplied via a PyNN script). All state variables are then updated one timestep at a time up to the simulation end time t_{end}. The number of required updates and hence *timer* events is calculated based on t_{end} and the user-defined simulation timestep Δt (which is fixed for the duration of simulation). Each call to `timer_callback` advances all the neurons on a core by Δt according to Algorithm S1 in [208], which is shown schematically on the left-hand side of Figure 4.18. First the synapse state for all neurons on the core is updated according to the model shaping rule, and any new input this timestep is added from the synaptic input buffers (discussed below). Interrupts are disabled during this update to prevent concurrent access to the buffers from spike processing operations. The states of all neurons on the core are then updated sequentially. An individual neuron state at the current time $N_{i,t}$ is accessed in memory, and if the neuron is not refractory, its state is updated according to the model characterising its sub-threshold

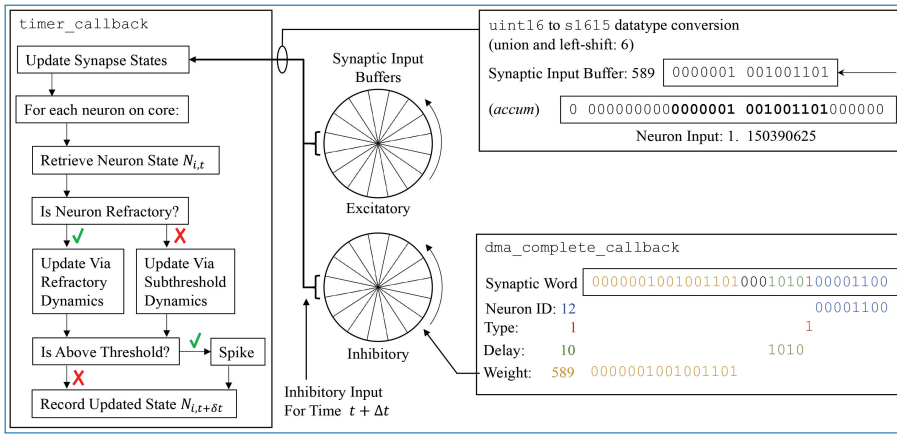

Figure 4.18. Left: update flow advancing state of neuron N_i by Δt. Centre: circular synaptic input buffers accumulate scaled input at different locations based on synaptic delay (buffers are rotated one slot at the end of every timestep). Right top: synaptic input buffer values are converted to fixed-point format and scaled before adding to N_i. Right bottom: decoding of synaptic word into circular synaptic buffer input.

dynamics (see examples in Section 4.9.5). If it is judged to have emitted a spike, the refractory dynamics are initiated and the router is instructed to send a multicast packet to the network. Finally, all requested neuron variables are recorded as belonging to this new timestep ($t + \Delta t$) and stored in core memory for subsequent extraction by the SpiNNTools software – interrupts are disabled during this process to prevent concurrent access to recording datastructures.

Synaptic input buffers (Figure 4.18 centre) are used to accumulate all synaptic input on a given receptor type, removing the computational cost of managing state variables for individual synapses (as developed by Morrison *et al.* [172]). Each buffer is constructed from a number of 'slots', where each slot represents input at a future simulation timestep. All input designated to arrive at a particular time is accumulated in the appropriate slot, constraining synapse models to those whose contributions can be summed linearly. A pointer is maintained to the input associated with the proceeding timestep ($t + \Delta t$). Each neuron update consumes the input addressed by this pointer and then advances it forward one slot (effectively rotating the buffer). When the pointer reaches the last slot, it cycles back to the first, meaning these slots continuously represent input over the next d timesteps, where d is the number of slots. By default the value of d is set via a 4-bit unsigned integer, enabling representation of delays up to 16 timesteps (however, Section 4.9.6 contains information on extending this delay). In the default sPyNNaker implementation, a synaptic input buffer is created per neuron, per receptor type, and is a collection of 16 slots each constructed from unsigned 16-bit integers. The use of

an integer representation reduces buffer size in DTCM and also the size of synaptic weights in SDRAM, relative to using standard 32-bit fixed-point *accum* type. However, it requires conversion to *accum* type for use in the neuron model calculations – as shown in Figure 4.18. This conversion is performed via a union and left-shift, the size of which represents a trade-off between headroom and precision. An example shift of 6 is shown, causing the smallest bit of the synaptic input buffer to represent $2^{-9} = 1.953125 \times 10^{-3}$, and the largest $2^{7} = 128$, in the *accum* type of the synapse state. Under extreme conditions, a buffer slot will saturate from concurrent spike activity, meaning the shift size should be increased. However, the shift is also intrinsic to the weight representation and affects precision, as all weights must be scaled by $2^{(15-shift)}$ before being written as integers to the synaptic matrices discussed in Section 4.9.4. For example, in Figure 4.18, a weight of 1.15 nA was converted to 589 on host during generation of synaptic data, but is returned as 1.150390625 nA when used during simulation (with a shift of 6). The shift value is currently calculated by the sPyNNaker toolchain to provide a balance between handling large weights, high fan-in and/or pre-synaptic firing rates, and maintaining precision – see the work by Albada *et al.* [3] where the theory leading to a usable closed-form probabilistic headroom mechanism is described in Equation 1.

Receiving a Spike

A _multicast_packet_received_callback is triggered by a *packet received* event, raised when a multicast packet arrives at the core. This callback is assigned highest priority (−1) and hence makes use of the FIQ thread and pre-empts all other core processing (see Figure 4.16(a)). This callback cannot be queued, and therefore, to prevent traffic backing up on the network, this callback is designed to execute quickly, and it simply extracts the source neuron ID (from the 32-bit key) and stores it in an input spike buffer for subsequent processing. Note that by default this buffer is 256 entries long, enabling queuing of 256 spikes simultaneously. The callback then checks for activity in the spike processing pipeline and registers a *user* event if inactive. Pseudo code for this callback is made available by Rhodes *et al.* [208].

Activation of the Spike Processing Pipeline

A user_callback callback is triggered by the *user* event registered in a Section 4.9.4 and kick-starts the spike processing pipeline. The callback locates in SDRAM the synaptic data associated with the spike ID and initiates its DMA transfer to DTCM for subsequent processing. Three core-specific data structures are used in this process: the *master population table*, *address list* and *synaptic matrix*. Use of these data structures is shown schematically in Figure 4.19, from the perspective of the core simulating the Excitatory A population in Figure 4.15(b), when receiving a spike from the Excitatory A population. The *master population table* is a lightweight list

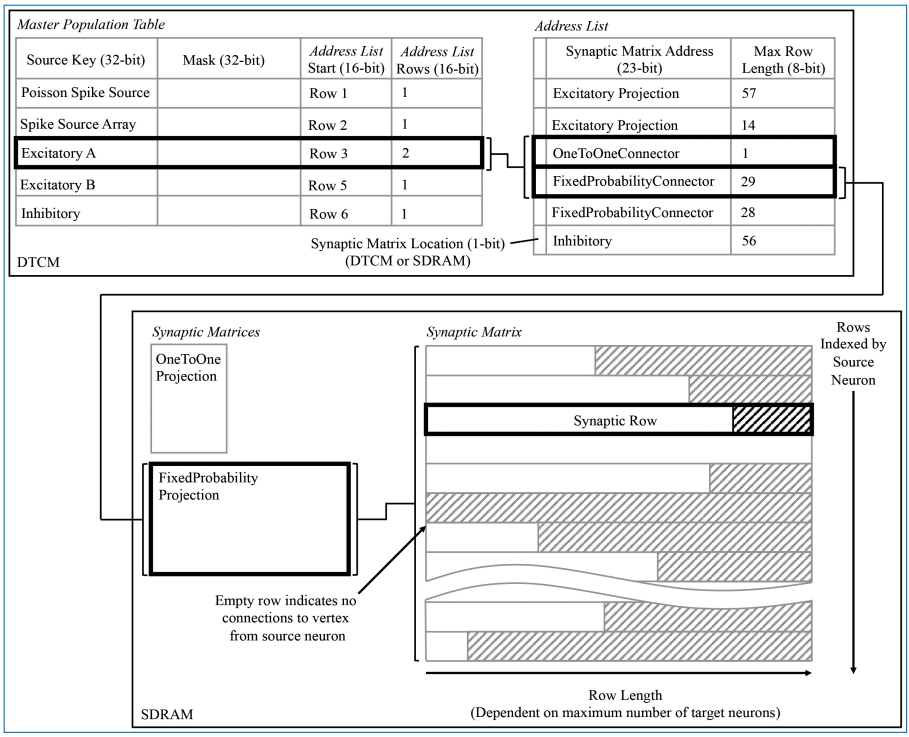

Figure 4.19. Data structures for processing incoming spikes: *Master population table*, *address list*, and *synaptic matrix*, are shown from the perspective of the core simulating the Excitatory A population in Figure 4.15(b). The path in bold represents that taken when a packet is received by Excitatory A, originating from itself, and hence two projections must be processed.

taking a masked source neuron ID as the key by which a source vertex can be identified. Each row pertains to a single source vertex and consists of: 32-bit key; 32-bit mask; 16-bit start location of the first row in the *address list* pertaining to this source vertex; and a 16-bit value defining the number of rows, where each row in the *address list* represents a PyNN projection. When searching this table, the key from the incoming packet is masked using each entry-specific mask before comparing to the entry key. This masks off the individual neuron ID bits and enables source vertices to simulate different numbers of neurons. The entry keys are masked on host before loading for efficiency and are structured to prevent overlap after masking and facilitate binary searching. The structure of an *address list* row consists of: a single header bit detailing whether the synaptic matrix associated with this projection is located in DTCM or SDRAM; 32-bit memory address indicating the first row of the synaptic matrix; and an 8-bit value detailing the synaptic matrix row length (i.e. the maximum number of post-synaptic neurons connected to by

a pre-synaptic neuron in a particular projection). Note that synaptic matrix rows are indexed by source neuron ID and that all rows are padded to the maximum row length to facilitate retrieval, including empty rows for pre-synaptic neurons not connected to neurons on this core. The row data structure is covered in detail in Section 4.9.4.

This callback therefore takes from the input spike buffer the next spike ID to process and uses it in a binary search of the *master population table* to locate the *address list* regions capturing the projections carrying the spike to this vertex. The SDRAM location and size specified by each row are then used in sequential processing of the projections. For the case shown in Figure 4.19, searching the *master population table* yields two rows in the *address list*, which in turn define the location of the corresponding synaptic matrices in SDRAM. Each synaptic matrix is indexed according to pre-synaptic neuron ID, enabling location of the appropriate row to copy to core DTCM for processing of each spike. Details of this row are then passed to the DMA controller to begin the data transfer, marking the end of the callback. This allows the core to return to processing other callbacks, hiding the DMA transfer as shown for 'Spike 1' in Figure 4.21.

Synapse processing

On completion of the DMA in Section 4.9.4, a *DMA complete* event triggers a `_dma_complete_callback`, initiating processing of the synaptic row. As described previously, each row pertains to synapses made, within a single PyNN projection, between a single pre-synaptic neuron and multiple post-synaptic neurons. At the highest level, a synaptic row is an array of synaptic words, where each word is defined as a 32-bit unsigned integer. The row is split into three designated regions to enable identification of static and plastic synapses (connections capable of changing their weight at runtime). The row regions contain dynamic plastic data, constant fixed plastic data and static data. Three header fields are also included, detailing the size of each region and enabling easy navigation of the row. A schematic breakdown of the synaptic row structure is detailed in Figure 4.20. Note that because a PyNN projection cannot be both static and plastic simultaneously, a single row contains only either static or plastic data. Plastic data are intentionally segregated into dynamic and fixed regions to facilitate processing. While all plastic data must be copied locally to evaluate synaptic contributions to a neuron, only the dynamic region – that is, that changing at runtime – requires updating for use when processing subsequent spikes. Keeping this dynamic data in a separate block facilitates writing back to the synaptic matrix with a single DMA, and writing back less data helps compensate for reduced DMA write bandwidth (relative to read – see Section 2.2).

The static region occupies the lower portion of the synaptic row and is itself an array of synaptic words, where each word corresponds to a synaptic connection

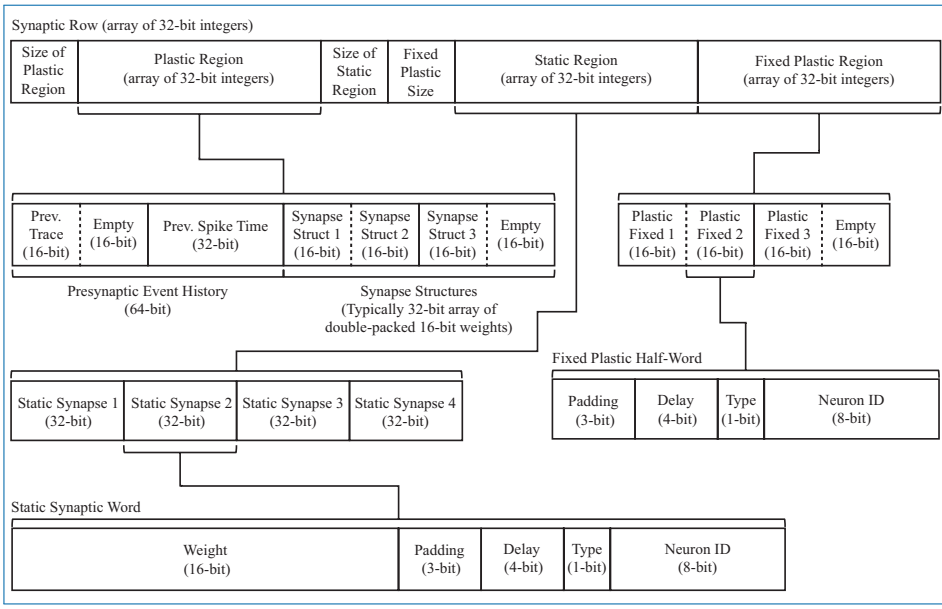

Figure 4.20. Synaptic row structure with breakdown of substructures for both static and plastic synapses.

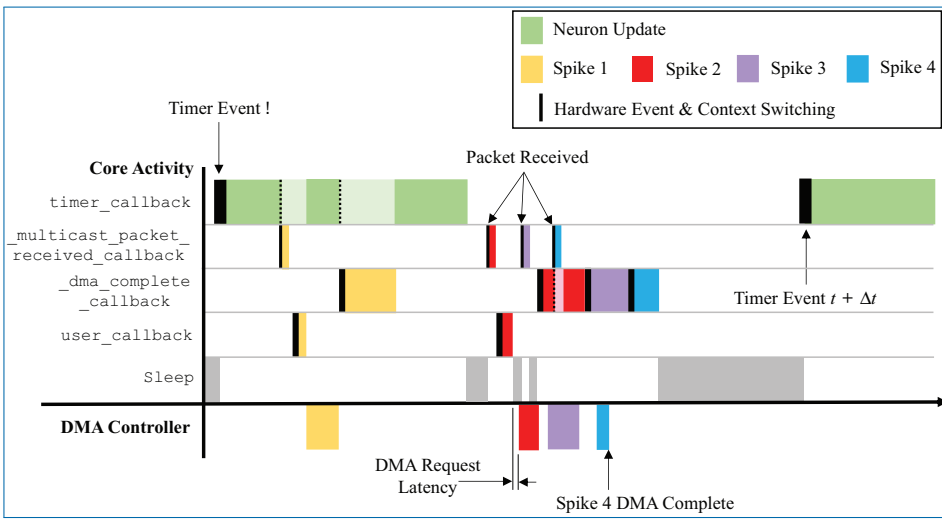

Figure 4.21. Interaction of callbacks shown over the time period between two *timer* events. Four spike events are processed representing the scenarios: receiving a packet while processing a timer event; receiving a packet while the core is idling; and receiving a packet while the spike processing pipeline is active. Note that a lighter colour shade indicates a suspension of a callback, which is resumed on completion of higher priority tasks.

between the row's pre-synaptic neuron and a single post-synaptic neuron. As shown in Figure 4.20, each 32-bit data structure is split such that the top 16 bits represent the weight, while the lower 16 bits typically split: bottom 8 bits to specify the post-synaptic neuron ID; 1 bit to specify the synapse type (excitatory 0, or inhibitory 1); 4 bits to specify synaptic delay; leaving 3 bits for padding (useful for model customisation, e.g., adding additional receptors types). Data defining plastic synapses are divided across the dynamic and fixed regions. Fixed plastic data are defined by a 16-bit unsigned integer and match the structure of the lower half of a static synapse (see lower half of Figure 4.20). These 16-bit synaptic half-words enable double-packing inside the 32-bit array of the synaptic row, meaning an empty half-slot will be apparent if the row targets an odd number of synapses. The dynamic plastic region contains a header defining the *pre-synaptic event history*, followed by a series of *synapse structures* capturing the weight of each synapse. Note that for typical plasticity models, this defaults to the same 16-bit weight describing static synapses; however, *synapse structure* can be extended to include additional parameters (in multiples of 16 bits) if required by a given plasticity rule.

A task of the _dma_complete_callback is therefore to convert the synaptic row into individual post-synaptic neuron input. The callback processes the row headers to ascertain whether it contains static or plastic data, adjusts synapses according to a given plasticity rule, and then loops over each synaptic word and extracts its neuronal contribution – pseudo code for this callback is detailed in Algorithm S4 of [208]. An example of this process for a single static synaptic word is shown in the lower right of Figure 4.18, where a synaptic word of [0000001001001101 0001010100001100] leads to a contribution of 589 to slot 10 of the inhibitory synaptic input buffer for neuron N_{12}.

Callback Interaction

The callbacks described above define how a sPyNNaker application responds to hardware events and updates an SNN simulation. The interaction of these events is a complex process, with the potential to impact the ability of a SpiNNaker machine to perform real-time execution. Figure 4.21 covers the time between two *timer* events and shows interaction of spike processing and neuron update callbacks for four scenarios detailed by the arrival of spikes 1–4. The first *timer* event initiates processing of the neuron update; however, after completion of approximately one-third of the update, the core receives Spike 1, interrupting the timer_callback and triggering execution of a _multicast_packet_received_callback, which in turn raises a *user* event, initiating DMA transfer of the appropriate synaptic information. On completion of the callback, the core returns to the timer_callback, with the DMA transfer occurring in parallel. On completion of the DMA, a

_dma_complete_callback is initiated, which processes the transferred synaptic information into neuronal input. The core then returns to the timer_callback, which continues to completion. The core is idle when it receives Spike 2; therefore, processing of the spike begins immediately, and the subsequent *user* event and hence DMA request is initiated. While waiting for the data to transfer, Spike 3 is received, and the associated _multicast_packet_received_callback is processed. This time, due to the active spike processing pipeline, no *user* event is raised, and instead, the DMA for Spike 3 is initiated at the beginning of the _dma_complete_callback triggered by Spike 2. Whilst processing this callback, Spike 4 is received, and the associated _multicast_packet_received_callback interrupts the core to place the packet key in the input spike queue. This queue entry is eventually processed at the beginning of the _dma_complete_callback for Spike 3, demonstrating the spike processing pipeline in action. This also shows the benefit of having two hardware 'threads' working in parallel, as the core is utilised completely, and the DMA transfer is hidden behind the _dma_complete_callback, when the pipeline is active. Finally, after an idle period (where the processor is put to sleep in a low energy state), the next *timer* event is issued at time $t + \Delta t$.

From Figure 4.21, it is seen that core processing is dependent on SNN activity. When targeting real-time execution (Section 4.9.4), it is important to consider extreme circumstances and how they will affect both the core and global simulation. For example, it is clear from Figure 4.21 that when a core receives spikes, it can delay completion of the timer_callback due to the assigned callback priorities (as shown in Figure 4.17). This is a design choice, as it helps maximise core utilisation by hiding DMA transfers behind the timer_callback when the spike processing pipeline is inactive. However, in the extreme case, spike processing will delay the completion of the callback beyond the issuing of the next *timer* event. While the core can potentially catch up this lost time, this scenario has the potential to delay the neuron update beyond a single *timer* event and ultimately cause any spike packets emitted from this core to be received and processed at the wrong time by the rest of the network. To guard against this, sPyNNaker applications report any occurrences of an overrun, where a timer_callback is not complete before the next *timer* event is raised and also the maximum number of *timer* events that a single timer_callback overruns. Similar metrics are also reported when the input spike queue overflows (exceeds 256 entries) and when the synaptic input buffers saturate. Together these metrics provide a window into the ability of a core to handle the required processing within a simulation.

Another important performance consideration when responding to spike packets using prioritised events is the time taken to switch between the associated

callbacks. Events are displayed in Figure 4.21 by solid black lines, the width of which represents the time taken to switch context and begin execution of the callback. The timer_callback takes longest to respond due to queuing of events with priority > 0, while the _multicast_packet_received_callback is quickest due to its priority of -1 and use of the FIQ thread. Other chip-level factors can also influence execution, such as SDRAM contention with applications running on adjacent cores. As DMAs are processed in serial bursts, if multiple simultaneous requests are received by the SDRAM controller, there may be latency in beginning the DMA for some cores and a reduced rate of transfer (see Section S1.2 of [208] for further information).

4.9.5 Neural Modelling

At the heart of a sPyNNaker application is the solution of a series of mathematical models governing neural dynamics. It is these models which determine how incoming spikes affect a neuron and when a neuron itself reaches threshold. While the preceding section described the underlying event-based operating system facilitating simulation and interaction of neurons, this section focuses on the solution of equations governing neural state and how they are structured in software.

Software Structure

PyNN defines a number of standard cell models, such as the LIF neuron and the Izhikevich neuron. Implementations of these standard models are included in sPyNNaker; however, the API is also designed to support users wishing to extend this core functionality and implement neuron models of their own. To facilitate this extension, the model framework is defined in an object-oriented fashion, through the use of C code on the SpiNNaker machine. This modular approach provides structure and aids code reuse between different models (e.g. sharing of a synaptic plasticity rule between different neuron models). A neuron model is built from the following components:

- *synapse_type*, defining how synapse state evolves between pre-synaptic spikes and how contributions from new spikes are added to the model. A fundamental requirement is that multiple synaptic inputs can be summed and shaped linearly, such as the α-kernel [49].
- *neuron_model*, implementing the sub-threshold update scheme and refractory dynamics.
- *input_type*, governing the process of converting synaptic input into neuron input current. Examples include current-based and conductance-based formulations [45].

- *threshold_type*, defining a system against which a neuron membrane potential is compared to adjudge whether a neuron has emitted a spike.
- *additional_input_type*, offering a flexible framework to model intrinsic currents dependent on the instantaneous membrane potential and potentially responding discontinuously on neuron firing (such as the Ca^{2+}-activated K^+ current described by Liu and Wang [149]).

The individual model components each produces a subset of the neuron and synapse dynamics and is therefore the entry point for a user looking to deploy a custom neuron model.[4] In keeping with the aforementioned software stacks in Figure 4.14, interfaces to each component are written in both Python and C. A single instance of each component is collected via a C header file and compiled against the underlying operating system described in Section 4.9.4 to generate a runtime application. Python classes for each component facilitate user interaction with each part of the model, enabling setting of parameter values and initial conditions from a PyNN SNN script.

The runtime execution framework calls each component as part of the timer_callback, as detailed in Algorithm S1 in [208] and shown schematically in Figure 4.18. First the synaptic state is advanced forward in time by a single simulation timestep, using the functions defined by the *synapse_type* component. Core interrupts are disabled during this process to prevent concurrent access of the synaptic input buffers from a _dma_complete_callback. Interrupts are re-enabled when all the state related to the synapses for all receptor types for all neurons on a core have been updated. Each neuron then has its state advanced by Δt. The *input_type* component is called first, converting the updated synaptic state into neuron input current. This includes separate excitatory and inhibitory components, with core implementations capable of handling both current- and conductance-based formulations. The *additional_input* component is then evaluated to calculate the level of any intrinsic currents. The synaptic and intrinsic currents, together with any background current, are then supplied to the *neuron_model* component which subsequently marches forward the neuron state by Δt. The neuron membrane potential is now passed to the *threshold_type* component which tests whether the neuron has fired. If the neuron is above threshold, a number of actions are performed: a refractory counter begins to instigate any refractory period; the *additional_input* is notified of the spike to allow updating of appropriate state variables; and finally, the core is instructed to send a multicast packet to the router with the neuron ID as key.

4. A detailed guide to this process can be found at: http://spinnakermanchester.github.io/workshops/seventh.html

Leaky Integrate and Fire Neuron

The sPyNNaker implementation of a current-based LIF neuron is described by the hybrid system in Equations 4.1 and 4.2. The sub-threshold dynamics are governed according to Equation 4.1, where V is the membrane potential, I is the input current (combining synaptic, intrinsic and background input), R_m is the membrane resistance, τ_m is the membrane leak time constant and E_l is the membrane leak (resting) potential.

$$\frac{dV}{dt} = -\frac{V - (E_l + R_m I(t))}{\tau_m} \qquad \text{if } V > V_\theta,\ V = V_{reset} \tag{4.1}$$

$$\frac{dI_{syn}}{dt} = -\frac{I_{syn}}{\tau_{syn}} + \delta(t - t^j) \tag{4.2}$$

If V exceeds the threshold level V_θ, the neuron is reported to have spiked and V is set to the reset potential V_{reset} for the refractory period duration t_r. Synaptic currents I_{syn} are modelled according to Equation 4.2, where τ_{syn} is the synaptic time constant (independent value for each receptor type), and the delta function represents addition of a step change in input from the weight of an incoming spike.

The sPyNNaker implementation embeds Equation 4.2 in a *synapse type* component, providing mechanisms to update the input current both between spikes (i.e. when the synaptic input buffer contribution is zero) and on spike arrival. Exact integration is used to update the synapse state during the periodic neuron update, with step changes made from synaptic input buffer contributions according to Equation 4.3.

$$I_{t+1} = I_t e^{-\Delta t/\tau_{syn}} + \Sigma_j w_{ij} \delta(t - t_j) \tag{4.3}$$

The constant factor $e^{-\Delta t/\tau_{syn}}$ is pre-calculated before loading to the SpiNNaker machine to avoid evaluation at runtime, as both the divide and exponential operations are relatively expensive on the ARM968 ($\approx$100 clock cycles each). A *neuron model* component captures the neuron state update mechanism, which solves Equation 4.1 via exponential integration [212] and assuming the change in current over the timestep is small [45], yielding the update function in Equation 4.4.

$$V_{t+1} = E_l + R_m I_{t+\Delta t} - e^{-\frac{\Delta t}{\tau_m}}(E_l + R_m I_{t+\Delta t} - V_t) \tag{4.4}$$

To compensate for this assumption, w_{ij} is decayed before adding to the synapse to ensure the total charge input to a neuron matches the exact solution [3]. Static thresholding defined via the *threshold type* compares the instantaneous membrane potential to the threshold level V_θ.

Izhikevich Neuron

The Izhikevich neuron model [116] allows reproduction of biologically observed neuronal characteristics such as spiking and bursting. Its dynamics follow a type of 'quadratic integrate and fire' model, as detailed in Equation 4.5

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I(t) \tag{4.5}$$
$$\frac{du}{dt} = a(bv - u)$$
$$\text{if } v \geq V_\theta, \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{4.6}$$

where v and u are dimensionless variables representing the membrane potential and a recovery variable, respectively. Dimensionless parameters a, b, c and d are used to tune the model dynamics, and I represents combined background, intrinsic and synaptic currents. If v exceeds a threshold V_θ, v and u are reset according to Equation 4.6.

The sPyNNaker implementation of this model uses the same *synapse type*, current-based *input type* and static *threshold type* components as the aforementioned LIF implementation. However, updating the neuron state and hence solving the system defined by Equation 4.5 requires numerical integration. A range of solvers were explored with fixed-point data type by Hopkins and Furber [101], with the RK-2 midpoint preferred as a trade-off between speed and accuracy. The resulting explicit update scheme is detailed in Equation 4.7.

$$\theta = 140 + I_{t+\Delta t} - u_t \qquad \alpha = \theta + (5 + 0.04v_t)v_t$$
$$\eta = \frac{\alpha h}{2} + v_t \qquad \beta = \frac{ah}{2}(bv_t - u_t)$$
$$v_{t+\Delta t} = v_t + h(\theta - \beta + (0.04\eta + 5)\eta)$$
$$u_{t+\Delta t} = u_t + ah(b\eta - \beta - u_t) \tag{4.7}$$

While it is hard to recognise the original equations in this form, refactoring of the update scheme and algebraic manipulation leads to several improvements in the implementation. The use of intermediate variables not only enables compiler optimisations improving speed and code size but also helps prevent over/underflow of the *accum* data type during intermediate calculations [101].

4.9.6 Auxiliary Application Code

While neuron-simulating applications capture the core operations of an SNN, several additional sPyNNaker applications are required to generate network input and facilitate network operation. These single-core applications are built following similar principles to those defined in Section 4.9.4, responding via the same event-based operating system to send and receive packets and interact with neuron cores. They are embedded in the machine graph during network preprocessing and loaded onto a SpiNNaker machine together with configuration data.

Spike Input Generation

Generating spikes is an integral part of SNN simulations. It enables modelling of network response to specific patterns of spikes and input representing adjacent brain regions or background noise. The sPyNNaker API includes two applications for spike generation: *spike source array* and *Poisson spike source*. These applications are built from compiled C and require a single SpiNNaker core per instance. They follow *timer* events in parallel (but asynchronously) with neuron-simulating cores and send multicast packets representing spikes as discussed previously. These applications do not receive spikes and hence have their functionality encoded entirely in callbacks registered against *timer* events. As with all sPyNNaker applications, a corresponding Python class enables construction of a spike generator in a PyNN script and allows configuration data to be specified and subsequently loaded to a SpiNNaker machine.

The *spike source array* application contains a population of neuron-like units which emit spikes at specific times (see Listing 4.2). The times and keys to emit are stored in SDRAM and only copied into local DTCM when required during execution. The buffer of times/keys is pre-loaded up to memory limits and can be replenished during execution by sending requests to the host, although this is limited by the bandwidth of the on-board Ethernet. Callbacks issued on *timer* events (corresponding to timestep updates on neuron cores) then send packets to the router at the prescribed times. If multiple 'neurons' are registered to emit spike packets over the same timestep, a small random delay is added between sending of the packets to reduce pressure on the router.

The *Poisson spike source* application emits packets according to a Poisson distribution about a given frequency. A population of neuron-like units is specified, each of which can be assigned an individual mean firing rate (see Listing 4.2). At runtime, periodic *timer* events trigger a callback at every simulation timestep Δt, which assesses whether the core should send a packet to the router representing a spike. A distinction is made between slow and fast Poisson spike sources based on whether they emit fewer > 1 spike for any Δt. For fast spike sources, the number of spikes to

send between *timer* events is calculated [130], and the corresponding packets sent are interspersed with random delays. This random spacing reduces the chance of synchronised spike arrival at post-synaptic cores, easing pressure on both the source and target routers. For slow sources, after each spike, an inter-spike interval is evaluated in multiples of Δt, which is then counted down between sending packets. For fast spike sources, the post-synaptic core is likely to retrieve from SDRAM the same pieces of synaptic matrix many times during a simulation. Therefore, to remove the overhead of the DMA, a mechanism is included to store the synaptic matrices from fast spike sources in DTCM.

Simulating Extended Synaptic Delays

While there is a mechanism in the synaptic row to account for delays of up to $16\Delta t$, it can be necessary to prescribe longer delays (particularly when Δ is small). To account for this case, an application called a *delay extension* is created [3], running on an adjacent core. Packets representing spikes exhibiting a delay $\geq 16\Delta t$ are routed to the core running this application, which subsequently sends new spikes targeting the post-synaptic core after a sufficient portion of the delay has elapsed such that any remaining delay can be handled within the synaptic row.

Two data structures are used to manage delay handling: a 'delay stage configuration' is generated during preprocessing and captures the size of delay associated with each pre-synaptic neuron; and a 'spike counter' registers the time and pre-synaptic neuron of incoming spikes. Two callbacks are used in the *delay extension*, registered against *packet received* and *timer* hardware events. On packet arrival, the first callback extracts the pre-synaptic neuron ID to an input spike buffer, similar to the process described in Section 4.9.4. The second callback is executed on *timer* events occurring in parallel (but asynchronously) with those on neuron processing cores. The callback processes any spikes received since the previous *timer* event, taking entries from the input spike buffer and using them to update the spike counting data structure to register the incoming spikes against multiples of the number of synaptic input buffer slots on the corresponding post-synaptic core. There are typically 16 such slots, where in this context a collection of 16 slots is referred to as a 'delay stage'. A second data structure captures how many delay stages each spike should be held for before being released to the post-synaptic core. Therefore, using these two data structures, it is possible to assess the incoming spikes to calculate the corresponding outgoing spike times and hence schedule the necessary spikes for distribution to the network.

While this application solves the problem of simulating extended delays, it cannot do so indefinitely and an effective new upper limit of $144\Delta t$ is enforced due to DTCM constraints. It should also be noted that this mechanism introduces additional overhead to the system: an extra core is required to run the application, and

two packets are now required to transmit a spike. The post-synaptic core also performs additional processing during look-up of the source vertex in the *master population table*. An additional row must be included to identify spikes travelling direct from the pre-synaptic core and also those sent from each individual delay stage of the *delay extension*. This increased *master population table* size can be costly to search and detrimental for real-time performance [207].

4.10 Software Engineering for Future Systems

As this text is being written, the SpiNNaker2 system, described in Chapter 8, is being developed. This architecture has clear implications on the software. To this end, it makes sense to develop the software to require as few changes as possible to make it compatible with this new system. The hierarchical modular structure of the software supports this well; for example, the mapping algorithms operate on a machine object, which can be simply updated when the structure of the new system is known (or a second version can be created and algorithms can operate on whichever system is in use). Similarly, the communications layer will require updating to match the communications used by the new system. However, the concepts will be similar to the higher levels, and so they will be able to stay the same, for example, the communications layer will have to support 'executing of a binary' and 'loading of data' but the signature of these functions can be made the same for both the old and the new system, avoiding the need to change the high-level libraries.

The other part of the system that would require changes is within the C code, where the features of the new system will need to be made accessible through the low-level libraries. Again, concepts that exist in both systems, such as the ability to run in an event-based manner, will map directly to the hardware, and so high-level code will not have to change (other than requiring recompilation of course).

All this means that minimal code changes will be required to make the code compatible with both old and new (and possibly even newer) systems. This makes the code somewhat future proof in so much as any software can be and require minimal maintenance as the hardware systems are developed.

4.11 Full Example Code Listing

```
import pyNN.spiNNaker as sim

# Initialise simulator
sim.setup(timestep=1)

# Spike input
poisson_spike_source = sim.Population(250, sim.SpikeSourcePoisson(
    rate=50, duration=5000), label='poisson_source')

spike_source_array = sim.Population(250, sim.SpikeSourceArray,
                                    {'spike_times': [1000]},
                                    label='spike_source')


# Neuron Parameters
cell_params_exc = {
    'tau_m': 20.0, 'cm': 1.0, 'v_rest': -65.0, 'v_reset': -65.0,
    'v_thresh': -50.0, 'tau_syn_E': 5.0, 'tau_syn_I': 15.0,
    'tau_refrac': 0.3, 'i_offset': 0}

cell_params_inh = {
    'tau_m': 20.0, 'cm': 1.0, 'v_rest': -65.0, 'v_reset': -65.0,
    'v_thresh': -50.0, 'tau_syn_E': 5.0, 'tau_syn_I': 5.0,
    'tau_refrac': 0.3, 'i_offset': 0}

# Neuronal populations
pop_exc = sim.Population(500, sim.IF_curr_exp(**cell_params_exc),
                         label='excitatory_pop')

pop_inh = sim.Population(125, sim.IF_curr_exp(**cell_params_inh),
                         label='inhibitory_pop')


# Generate random distributions from which to initialise parameters
rng = sim.NumpyRNG(seed=98766987, parallel_safe=True)

# Initialise membrane potentials uniformly between threshold and resting
pop_exc.set(v=sim.RandomDistribution('uniform',
                                     [cell_params_exc['v_reset'],
                                      cell_params_exc['v_thresh']],
                                      rng=rng))

# Distribution from which to allocate delays
delay_distribution = sim.RandomDistribution('uniform', [1, 10], rng=rng)

# Spike input projections
spike_source_projection = sim.Projection(spike_source_array, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.05),
    synapse_type=sim.StaticSynapse(weight=0.1, delay=delay_distribution),
    receptor_type='excitatory')

# Poisson source projections
poisson_projection_exc = sim.Projection(poisson_spike_source, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.06, delay=delay_distribution),
    receptor_type='excitatory')
poisson_projection_inh = sim.Projection(poisson_spike_source, pop_inh,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.03, delay=delay_distribution),
    receptor_type='excitatory')
```

```
# Recurrent projections
exc_exc_rec = sim.Projection(pop_exc, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.1),
    synapse_type=sim.StaticSynapse(weight=0.03, delay=delay_distribution),
    receptor_type='excitatory')
exc_exc_one_to_one_rec = sim.Projection(pop_exc, pop_exc,
    sim.OneToOneConnector(),
    synapse_type=sim.StaticSynapse(weight=0.03, delay=delay_distribution),
    receptor_type='excitatory')
inh_inh_rec = sim.Projection(pop_inh, pop_inh,
    sim.FixedProbabilityConnector(p_connect=0.1),
    synapse_type=sim.StaticSynapse(weight=0.03, delay=delay_distribution),
    receptor_type='inhibitory')

# Projections between neuronal populations
exc_to_inh = sim.Projection(pop_exc, pop_inh,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.06, delay=delay_distribution),
    receptor_type='excitatory')
inh_to_exc = sim.Projection(pop_inh, pop_exc,
    sim.FixedProbabilityConnector(p_connect=0.2),
    synapse_type=sim.StaticSynapse(weight=0.06, delay=delay_distribution),
    receptor_type='inhibitory')


# Specify output recording
pop_exc.record('all')
pop_inh.record('spikes')


# Run simulation
sim.run(simtime=5000)


# Extract results data
exc_data = pop_exc.get_data('spikes')
inh_data = pop_inh.get_data('spikes')


# Exit simulation
sim.end()
```

Listing 4.2. An example for PyNN commands.