Flexible on-line reconfiguration of multi-core neuromorphic platforms

Flexible on-line reconfiguration of multi-core neuromorphic platforms

Francesco Barchi, *Student Member, IEEE,* Gianvito Urgese *Member, IEEE*, Alessandro Siino, Santa Di Cataldo, Enrico Macii, *Fellow Member, IEEE* and Andrea Acquaviva, *Member, IEEE*

Abstract—Neuromorphic architectures are emerging not only for real-time simulation of brain-scale biological neural networks but also to support innovative brain-inspired computational paradigms. In both domains there is an increasing demand for flexibility in terms of network configuration and runtime redesign of network parameters and simulated neurons models. Due to the intrinsically high parallelism of these architectures and complexity of the interconnect, broadcasting updates to the cores is time consuming. Hence, static solutions where the network is reloaded from an external host instead of being reconfigured are highly inefficient. To address these requirements, we designed an *Application Command Protocol (ACP).* The proposed protocol provides a mechanism to remotely trigger the execution of high-level op-codes by the cores and manage their application memory, and supports a more flexible computational model and memory management. We worked on SpiNNaker, a multi-core globally-asynchronous locally-synchronous platform running Spiking Neural Networks (SNNs) simulations. We demonstrated ACP in two SNN applications: i) SNN configuration, where simulation data are efficiently generated through ACP in the memory of computing nodes and ii) SNN reconfiguration, where ACP is used to change SNN network parameters at runtime and to easily switch from learning to test phase in a SNN classification application.

1 Introduction

Neuromorphic platforms represent an intensive research area because of their capability of efficiently simulating biologically-plausible neural networks. Neuromorphic platforms are highly parallel architectures with a dense interconnect which allows to efficiently communicate signals (i.e. spikes) between computing nodes, to mimic brain neural connections.

Their use in computational neuroscience aims at studying biological interactions underlying brain functions through the simulation of neural networks using physical neuron models [1], [2], [3], [4].

Spiking neural networks (SNNs) or *third generation* neural networks [5] are commonly adopted in this context because they exploit Ordinary Differential Equations describing the behaviour of neurons. In the presence of a stimulus, each neuron communicates with other neurons through electric pulses (*spikes*). A spike generated by a neuron travels to other neurons and increases or decreases their potentials according to connection weights (*synapses*).

Because of their effectiveness in executing tasks in which the human brain is typically much more efficient compared to more traditional computation models, neuromorphic platforms are attracting ever-growing interest as hardware support for innovative brain-inspired computational paradigms [6]. Because of their inherent capability of using the temporal dimension (i.e. the time interval between spikes) to encode information, SNNs are successfully applied to various domains where temporal dynamics are relevant, such as speech and image processing, data-stream classification, and autonomous robotics [5], [7], [8].

A wealth of research is dedicated to improving various aspects of neuromorphic platforms, such as processing time, power consumption and flexibility of the computation model. In this paper we focus on the latter, trying to overcome the limitations impacting both brain simulation efficiency as well as their usage as accelerators.

In both domains (SNN simulation and SNN applications), these platforms have to be loaded with data to configure the network. Typically these platforms are connected to an external host computer where configuration information is generated. A host-to-platform communication channel is used to convey this data to an input port of the platform. From here, data should be broadcasted to the computing nodes simulating the neuron models. Due to the high degree of parallelism and the complexity of the interconnect, this communication is in general time consuming and needs to be efficiently managed.

This aspect becomes critical not only during initial network configuration time but also when several simulations have to be performed to explore network parameters in brain simulations, where configuration data must be reloaded. Moreover, in SNN applications it may be needed to update neuron models. For instance, the host must trigger a switch from a "learning" neuron model used during network training to a "test" neuron model during a classification task execution. This would require a communication protocol supporting a host-controlled runtime network update. In general, these platforms would benefit from a mechanism supporting remotely triggered self-reconfiguration, to reduce the cost of communication from the external host and to better exploit their inherent parallelism. Also, implementing a host control of cores execution flow, their usage as accelerators would be more effective and flexible.

- *All authors are with the Department of Control and Computer Engineering - DAUIN, Politecnico di Torino, Torino 10129, Italy*
  E-mail: gianvito.urgese@polito.it
- *This research has been founded by Human Brain Project.*

Manuscript received MONTH dd, yyyy; revised MONTH dd, yyyy.

In this paper, we describe the *Application Command Protocol (ACP)* that we designed for this purpose. We implemented it on a general-purpose many-core neuromorphic platform called SpiNNaker [4] (Figure 1). ACP works at the application level and extends the state-of-the-art software for this platform increasing its flexibility and efficiency of reconfiguration. In particular, it allows the transmission and interpretation of high-level op-codes defined by the users and embedded in the distributed applications (i.e. *Remote Procedure Calls - RPCs*). Indeed, cores can thus execute commands transmitted by the host. The protocol supports both host-to-platform and core-to-core communication. Through this mechanism, the protocol also implements the possibility to manage the cores memory (i.e. triggering read/write operations) by abstracting physical memory addresses using virtual IDs (defined as *Memory Entities*). As a result, cores can communicate, trigger operations and synchronise their execution.

We demonstrate the effectiveness of the proposed protocol in two applications. One is the interactive data loading, where we use ACP to generate SNN configuration data interactively on-board, meaning that host packets transmission and configuration data generation phases can be overlapped. This reduces memory requirements and better exploits the platform parallelism.

The second application is the SNN application reconfiguration. Here ACP allows the host to change the synapse delay during SNN simulation or to update the neuron model from learning to test in an SNN classification task.

Besides these examples, more generally ACP enables the embedding of alternative computational flows in the applications running on the board allowing the host to control their behaviour at runtime through RPCs and manage their memory using Memory Entities. Exploiting these features, ACP also provides core-to-core communication and synchronisation support.

The rest of the paper is organised as follows. Section 2 provides an overview of the neuromorphic platform used in this work, with a description of the SW/HW architecture. Section 3 introduces the main concepts of the ACP protocol, while Section 4 describes ACP design and implementation features in details. Section 5 presents ACP protocol performance tests and results obtained when applying the protocol in the two SNN applications case studies. Section 6 draws conclusion and highlights future developments.

2 Background

SpiNNaker is an application-specific massively parallel architecture designed in 2006 by Furber et al. [4] for simulating a large scale SNNs in real-time. Platform configuration requires several software modules for converting the SNNs simulations, designed by neuroscientists, into executable and configuration files to set-up the board [9], [10].

The SpiNNaker HW architecture is made of general purpose ARM cores. These processors are flexible and capable of aiding in the rapid evolution of neuroscience research.

The main features of the SpiNNaker platform are reported in the following subsections. Further details about the SpiNNaker architecture can be found in [4], [11]. Supporting tools are described in [12] while Rast et al. [13], [14] describe spinnaker communication protocols and systems. In [15] is described the profiling methodology adopted to highlight some of the bottlenecks of the SpiNNaker communication system. Whereas in [16], [17] authors introduce a new partitioning and placement algorithm developed to place the neurons to be simulated on the SpiNNaker board evaluating the impact of these techniques on the reliability of the simulations.

2.1 An overview on the SpiNNaker Machine

A SpiNNaker Machine is a massively-parallel architecture (1 036 800 SpiNNaker Cores) designed to simulate in real-time Spiking Neural Networks (SNNs). The simulation begins in the host computer, where SNNs are described in software using PyNN [18], a simulator-independent Python library. The PyNN description of the SNN is elaborated in the host using a set of Python packages able to translate the SNN into configuration files to be sent to the SpiNNaker board [16], [19], [20]. Then, in the standard configuration flow, the generated files with detailed description

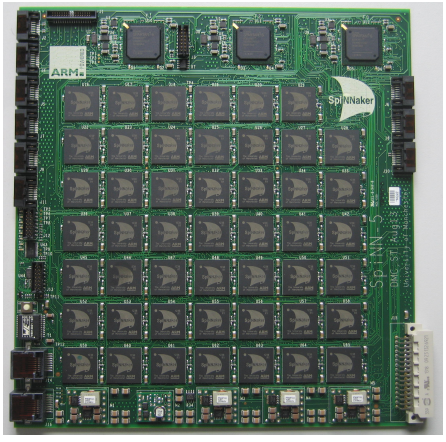

Figure 1: **The SpiNNaker Board**: A building block of a SpiNNaker machine, containing 48 chips for a total of 864 ARM processors.

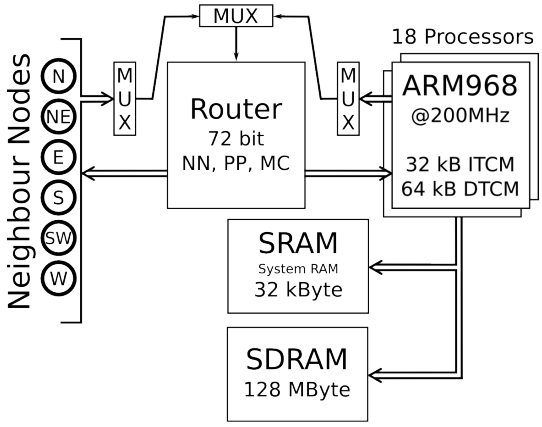

Figure 2: **SpiNNaker Architecture**: The schematics of a SpiNNaker Chip with processors, router and shared memory.

of memory regions are transmitted to every single core of the SpiNNaker Board involved in the simulation. After this configuration phase, the simulation can start. During the execution of SNN simulations on the SpiNNaker Machine, spikes are represented as packets and transmitted through the network using the routers of the SpiNNaker chips [21]. A minimal operating system kernel (called SARK) and a set of light libraries are used by the cores to support the execution of the code and to manage the in/out packet transmission.

The primary element of the SpiNNaker Machine is the SpiNNaker Chip that represents a single node of the system [13]. This node, shown in Figure 2, is a SoC comprising 18 SpiNNaker Cores, 32 kB of System SRAM, 128 MB of DDR-SDRAM and a custom router. Each SpiNNaker Core is an ARMv5TE processor (ARM968@200 MHz) equipped with two Tightly Coupled Memories: 32 kB for Instructions and 64 kB for Data.

At the board start-up, each node selects a SpiNNaker Core to be used as *Monitor Processor (MP)* for managing the entire node. All the other 17 cores are available for the execution of user-defined applications written in C [10]. This last class of cores are called *Application Processors (APs)*.

The SpiNNaker Chips are welded on a SpiNNaker Board (SpiNN-5), a Printed Circuit Board that hosts 48 nodes connected in a hexagonal mesh. This network structure allows each chip to communicate with its six neighbours by using custom routers placed on each node, implementing an onboard communication system [22].

The off-board communication between the Host Computer and the SpiNNaker Board is supported using a 100 Mbit Ethernet interface connected to a single SpiNNaker Chip (Root Node). The Root Node that receives UDP packets from the Host is in charge of forwarding the payload of UDPs within the network of nodes.

2.2 Communication system

The SpiNNaker Machine has a multilayer communication system which uses the On-board routers, designed to support four types of packet, for implementing different communication protocols useful for low-level communication, kernel setting-up, host-to-board communication and (as we will see in 3 Section) to support the communication of parallel applications.

2.2.1 On-board communications

The custom router has a central role in the management of communications between nodes. The Router can transmit four types of packet, everyone not larger than 72 bits:

- The ***Point to Point (P2P)*** packet format is used by SpiNNaker Chips to transmit packets between two specific routers of the system (40 bits of header and 32 bits of payload). Only Monitor Processors can handle this type of packet.
- The ***Multicast (MC)*** format is used by Application Processors to transmit packets to many cores across the simulation. It is widely used in neural simulations to spread spikes to multiple destinations.
- The ***Nearest Neighbour (NN)*** packets are used at low-level for implementing a keep-alive mechanism to check for broken links. This type of packet can reach only the six neighbouring chips.
- The ***Fixed Route (FR)*** packet can be used by all the cores to exchange data with the Ethernet controller.

During the SNN simulations, the application processors use the multicast packets directly for transmitting the generated spike events to the target APs. On top that, a higher level protocol (*Spinnaker Datagram Protocol, SDP*) implemented in software is available for the transmission of larger payloads between cores. SDP is currently based on P2P communication.

2.2.2 Off-board communications

The communications with the Host Computer are made possible through the Ethernet interface connected to the Root Node, that redirects all the received *Spinnaker Datagram Protocol (SDP)* packets (encapsulated in UDP/IP) to its Monitor Processors [23] (Root Processor). The Root Processor, indeed, acts as a middleware between the SDP protocol and the network of nodes. Once the Root Processor retrieves an SDP packet, it splits the packet into 32-bit fragments and sends each fragment to the destination core through P2P packets. A mechanism of acknowledgements (ACK) between the Monitor Processor (MP) of the sender and the receiver MP makes the transmission of SDP packets reliable. When all the fragments of the whole SDP packet are received, the receiver MP copies the reconstructed SDP packet into a portion of System RAM (*Message Box*) and triggers an interrupt to the target core. Then, the target core can react to the interrupt and read the SDP from *Message Box*.

2.2.3 Kernel communications

The most used protocol for interacting with the OS kernel running on cores is the *Spinnaker Command Protocol (SCP)*, which is a format encapsulated in the data field of the SDP packet [24]. The header comprises 16 Bytes, and the supported payload is up to 256 Bytes. SCP is used for low-level interactions with SpiNNaker systems such as simulation configuration, program loading, and debugging. Kernel software running on every active core accepts the commands and use them to perform low-level functions such as getting kernel version running on the SpiNNaker Cores, reading/writing the memory locations, and triggering the execution of programs. The SCP provides four low-level instructions for accessing chip resources and extracting debugging information, such as the working state of the APs or the number of packets processed by the Router. Furthermore, SCP provides signals for controlling the application execution state and for modifying the AP memory at low-level.

The implementation of this protocol is embedded in the kernel (SARK) that must be kept as light as possible because of the limited memory resources and small computational power of the cores. Working at the kernel level and missing a direct interface to applications, SCP cannot be used to support the features discussed in this work, for which we need an application-level protocol such as ACP.

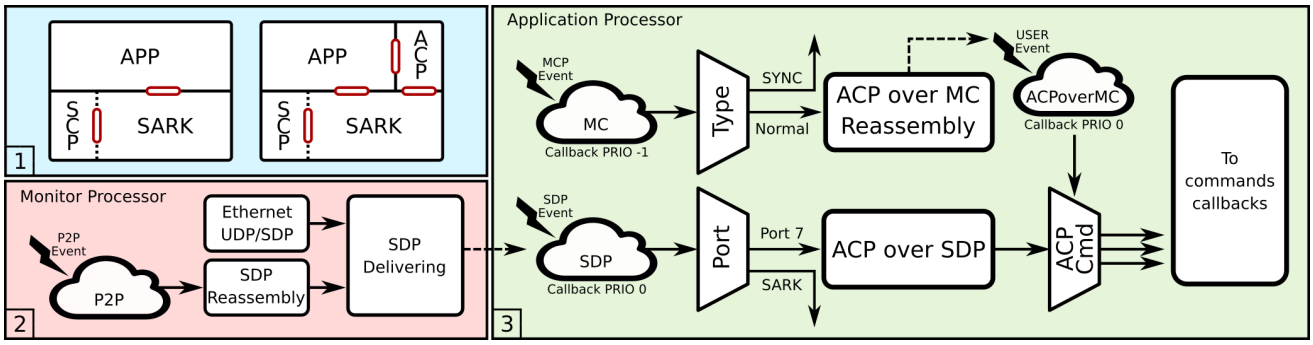

Figure 3: **ACP framework and transmission flow.** In box 1, the ACP framework interactions. In box 2, the Monitor Processor workflow for managing the incoming packets. In box 3, the Application Processor workflow performed to interpret the incoming commands.

3 The Application Command Protocol

In this section we define the *Application Command Protocol* (ACP), a high-level command protocol and memory management system that can be used to reconfigure memory data structures of the application software running on SpiNNaker.

The first section of Figure 3 (Figure 3.1) depicts a simplified representation of the software stack. More specifically, the current version of the stack without ACP is shown on the left, and our proposed solution leveraging APC on the right. Interfaces are also highlighted in the figure.

As can be gathered from the upper left section of the figure, there is no direct interface between SCP and applications. Hence, using SCP to interact with applications requires passing through significant modifications of the kernel. On the other hand, in our proposed solution ACP provides direct communication with the application. This significantly reduces the development efforts and improves the usability and flexibility of the software stack. Based on this design, we could implement the new RPC and memory management functionalities (based on Memory Entities) provided by ACP.

We implemented ACP as an application library that can be used by the applications.

To support the communication between processors in distributed applications, we defined the ACP packet format. The header of an ACP packet consists of 4 Bytes containing the *Packet ID* and the *Command ID*. The following bytes are used for the header of the command (from 4 to 12 Bytes) depending on which *command ID* is in the packet. Likewise, the format of the *Command Payload* depends on the type of command, it has a maximum limit of 264 Bytes length.

An ACP message can be transmitted over the board using two communication channels. The *ACP over SDP* uses the SDP protocol implemented in the native SpiNNaker software stack. The Monitor Processor of the sender chip breaks the SDP packet into P2P packets and sends them to the Monitor Processor of the receiving chip. The *ACP over MC* uses the multicast (MC) channel. The sender Application Processor breaks each ACP packet into a set of MC packets, each one is a 32 bits fragment, and transmits them using the SpiNNaker native multicast transmission protocol. In a recent work [25], we have used the MC channel for implementing minimal support of an MPI implementation and a synchronisation system for the SpiNNaker platform.

4 ACP workflow

In Figures 3.2, 3.3 we show the workflow designed to manage the ACP. The Monitor Processor mediates the *ACP over SDP* implementation using point to point connections. Whereas the *ACP over MC* implementation is directly managed by the Application Processors and allows for multicast[1] and broadcast[2] transmissions.

The ACP is implemented in two libraries, *SpynnakerACP* and *SpinACP*. The *SpynnakerACP* is implemented as a Python package and is organised as a collection of classes and utility methods used in the host computer to create, send and receive commands to SpiNNaker chips through the Ethernet connection. The library provides a framework to define the functions to be implemented when a SpiNNaker core receives a command. The framework is customisable, as users with particular needs can extend the default set of commands for supporting new functions that will help the design of flexible applications.

The *SpinACP* is built on top of SARK and uses the event-driven programming model provided by the *Spin1* library. This module provides three main functionalities: i) At network level, it implements the two systems for sending command packets over the network (*ACP over SDP* and *ACP over MC*). ii) It provides a customizable framework for supporting command management (*Remote Procedure Call*). iii) It implements an abstraction level of the memory blocks through the definition of *Memory Entities*.

4.1 Network Management

ACP allows two or more APs to communicate with each other. It also puts in communication an AP with the host giving the possibility to an AP to start a communication even if not explicitly interrogated by the host. Another innovation is the possibility of connecting and synchronising many APs with each other, using an integrated system that we have already exploited in MPI to synchronise the executed application globally.

1. One processor sends a message to a subset of processors.
2. One processor sends a message to all the others processors.

As discussed, ACP makes use of two types of packets (*ACP over SDP* and *ACP over MC*) to support the communication of a distributed application. When an application processor (AP_a) needs to send an ACP packet using the *ACP over SDP* channel to another application processor (AP_b), a sequence of events occur:

1) The AP_a writes the SDP inside the shared memory.
2) The AP_a sends an interrupt to its monitor processor (MP_a) to inform him of the presence of a pending packet.
3) The MP_a manages the request and fragments the packet into 32 bit chunks.
4) The MP_a sends the fragments to the monitor processor of the destination chip (MP_b) through the P2P channel.
5) The MP_b receives the P2P packets and rebuilds the SDP packet. When the SDP packet has been reconstructed, the MP_b saves it in the shared memory.
6) The MP_b sends an interrupt to the destination AP_b to inform him of the presence of a pending packet.
7) The AP_b retrieves the SDP message from the shared memory.

The ACP implementation intercepts the incoming SDP and checks the port field contained in its header. The port field is a 3-bit field and behaves as the TCP or UDP port. If the port field is set to $0b111$, the SDP payload is identified and passed to the ACP packet interpreter.

The ACP packet interpreter reads the *command ID* field and checks whether the command can be executed (i.e. its execution function is registered). If this is the case, command header and payload are passed to the next phase that is performed by the function associated with the command.

When an application processor needs to send an *ACP over SDP* to the host computer, the flow is similar, the only difference being that the MP_b is the Root Processor. The Root Processor envelopes the SDP inside a UDP/IP and then forwards it to the host computer, where the ACP runtime library is ready to intercept it. When the commands are generated off-board, e.g. by the host computer, they are transmitted to the Root Processor using UDP/IP. The Root Processor extracts the SDP packet and forwards it using the mechanism described above.

When an ACP command is sent to many destinations, it can be managed with *ACP over MC*. This mechanism, which is our specific design, does not make use of the Monitor Processors. Instead, the fragmentation and re-composition of the ACP into the MC packets is performed directly by the Application Processors by using a low-level protocol based on multicast packets [25].

With these two mechanisms, the ACP commands can be employed to change at run-time the execution flow and the data structures of the applications running on the APs. This feature can be exploited to trigger alternative execution flows, based on signals that are *not predictable* at configuration time.

4.2 Command Management

The command management ACP functionality (RPC) can be recreated in the SCP by modifying the SARK kernel in order to support new commands[3]. However, extending the available commands requires extensive kernel modifications, with additional efforts devoted to maintaining the kernel light, stable, and safe.

Instead, ACP only requires to create and record new call-back in the command call-back list, allowing the user to easily define custom commands. More specifically, the user registers the function implementing the command interpreter as a callback function on the ACP library. This function is associated with an identifier, *Command ID*, and stored in a hash-table. This solution provides much better flexibility than a static vector.

When the ACP packet (Figure 3.3) is processed, the following steps are involved:

1) The ACP header is read, and the Command ID is extracted.
2) The ACP library searches the hash table for the callback function coupled with the Command ID.
3) If the callback does not exist the command is ignored, otherwise the function pointer is extracted.
4) The selected callback reads the command header section and executes the command.
5) If required, the selected callback reads the command payload section.
6) If the command requires a reply, an ACP packet is created and appended to the outcome packet queue.

4.3 Memory entities

The ACP implementation also provides a set of built-in commands useful to manage *Memory Entities*. A memory entity is a block of bytes with a configurable size[4], that may be physically allocated in the DTCM or the SDRAM. A memory entity has an identifier, *Variable ID*, and it is internally described by an Abstract Data Type (ADT). Memory entity descriptors are collected (like the commands callbacks) inside a hash-table.

The life of a Memory Entity starts with its declaration specifying its *Variable ID* and its size. Furthermore, it is possible to associate an action, *callback*, to be executed when a memory entity is accessed. A memory entity callback can be registered within three different decorators. The *action decorator* defines which action triggers the callback: read, write or delete. The *temporal decorator* defines when the callback has to be triggered: before or after the command execution. The *provenance decorator* defines where the callback has to be triggered: if the command comes from the same processor (local command) or another processor (remote command).

This memory management system virtualises the memory block and allows much higher flexibility in memory access because it does not require access to the AP memory image saved in the Host Computer. In the current version, the ACP protocol does not provide a native retransmission system to prevent the loss of packets. However, users can

3. Currently each Application Processor can execute four SCP commands: Get Kernel Version, Memory Read, Memory Write and Application Run.

4. The maximum size is 256 Byte, like the maximum payload carried by an ACP packet

register a global *callback*, after the ACP header interpretation, to handle an unexpected Packet ID and to implement a system capable of recovering eventually lost packets.

5 ACP Case Studies

In order to demonstrate the advantages of the proposed solution, we exploited ACP to enhance existing applications in the development environment for SpiNNaker (Figure 4). Within such applications, we assessed the added value provided by ACP in terms of additional features compared to the existing support (i.e. SCP).

The first application taken into consideration is the program used to configure the cores before a simulation. We have introduced ACP in the configurator in order to overcome some limitations imposed by the current SCP-based system. The configurator consists of an interpreter of commands executed on each processor of the architecture, that needs a set of op-codes preloaded in memory. In the current system, the entire op-code sequence is introduced into memory using SCP. With ACP, a configurator is now capable of receiving op-codes at runtime, with a two-fold advantage: i) a reduced use of memory (it is not necessary to store an entire list of op-codes but only a portion) and ii) the parallelisation of the procedure (while sending op-codes to processor B, processor A starts to process its set of op-codes). This application has also been used to profile the performance of the framework.

The second application taken into account is the neuronal model used in SNN simulations. More specifically, we introduced ACP in the implementation of the neuronal model, in order to allow the reconfiguration of some operating parameters of the synapses during the simulation. This was exploited within two different applications: an SNN classifier, and an SNN simulation where we tune the synapse delay parameter.

The classifier SNN is made of two phases, learning and testing. The current SCP-based pipeline requires running a complete simulation in both phases, due to the necessity of using two different neural models, one for the learning and the other for the testing. The introduction of ACP avoids this overhead, allowing a re-configuration of the neuron's behaviour that makes it usable in both phases.

The SNN simulation consists of a linear sequence of neurons that stimulate each other. Within this application, we evaluated the possibility of either tuning SNN parameters or introducing complex behaviours to the simulation in real-time. More specifically, we introduced ACP in the neural model to allow the modification of the delay of the synapses during the simulation.

5.1 ACP for Interactive Data Loading

The benchmark application described in this section is a program used to configure SNN simulations. Our modified configurator runs on the host computer and sends commands to the SpiNNaker Board.

When a simulation runs, one of the very first steps involving the nodes of the board is the *data specification* (DS) phase. DS is one of the most critical phases concerning the time of execution as well as resources management. This

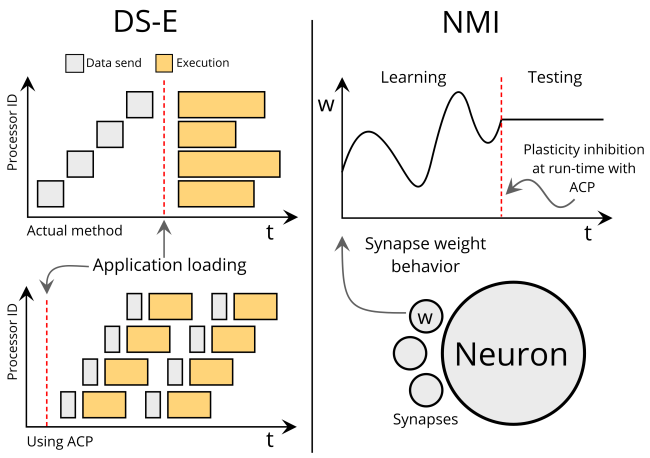

Figure 4: **Two application improved by the usage of ACP.** On the left, we used ACP for implementing a parallel transmission system of op-codes that configure the many processors of the board. On the right, we used ACP for triggering two different synaptic behaviours of a neuron model during an SNN simulation.

phase aims to fill the memory of Application Processors (APs) with the configuration data needed for running a simulation. Two implementations of the DS phase were discussed in previous work by Siino et al. [26]. In the following, we briefly summarise the main features.

Each NMI is equipped with its *data specification generator* (DS-G). The DS-G produces a sequence of commands (op-codes) that together make up the *data specification program* (DS-P). The DS-P is executed by a virtual machine called *data specification executor* (DS-E) which, processing each op-code, configures the memory of the application processor.

The DS-E can be performed on the host computer (*DS-E on-host*) or directly on SpiNNaker Board (*DS-E on-board*).

The DS-E on-host version produces a memory image for each AP. All data are sent to the SpiNNaker Board and written in the memory of each involved core. As the full memory image is transmitted in a serial way core-by-core, this implementation does not fully exploit the intrinsic high-parallelism and low-power consumption of the SpiNNaker system. The computational effort of DS-G and DS-E phases remains on the host side.

A DS-E on-board version is also available and can be executed by SpiNNaker application processors and deployed into each core involved in the simulation. To use this version, the host must first transfer DS-P to the cores. Then, the on-board DS-E will generate the data structures directly on the SpiNNaker memory.

While this implementation avoids the need to transfer core application memory images from the host, still it requires to upload the full DS-P to the core memories. With ACP, we overcome this limitation, and we provide two advantages: i) The DS-P can be executed on-the-fly through RPCs sent by the host (*Interactive DS-E On-board*), without requiring its complete transfer, thus saving memory; ii) It allows to exploit platform parallelism, as configuration commands are spread to the cores that can generate their data structures in parallel (*Interleaved Transmission*).

5.1.1 Interactive DS-E On-board

The *interactive DS-E on-board* makes use of the ACP for the transmission of the *data specification program*. We use this application to evaluate the performance and the reliability of the ACP implementation counting the number of packets lost as a function of the *Packet Delivery Delay Time* (t_{pdd}), that is the time elapsed between the transmission of two packets. This delay between packets avoids overloading the monitor processors involved in the transmission, thus improving the reliability of host-to-board data transmission.

We used as *testcase* the configuration of a biologically inspired SNN designed by Potjans et al. [27] and implementing the four layers constituting the human brain Cortical Microcircuit (CM). We scaled-down the number of neurons and synapses to 10% of the original network (CM_{10}) to satisfy time and resources constraints for fitting the SNN simulation in a single SpiNN-5. The SpiNNaker software maps the CM_{10} SNN on 240 cores distributed among 15 chips of a single SpiNNaker Board.

The Figure 5 reports the per-processor distribution of memory requirements of this application when SCP is used to load the whole data specification programs. As it can be easily gathered from the plot, the distribution of DS-P size is very heterogeneous, and a significant amount of the overall data transmission have very large memory requirements (>1 MB). On the other hand, using ACP for the same application, we obtained data transmission always happening in chunks of small fixed size (1 kB). Hence, ACP reduces the memory footprint by 90% at least. This has a positive impact on the memory access time, as it allows to leverage the fast DTCM memory (64 kB upper-bound). On top of that using the ACP the overall CM_{10} configuration time is reduced from 213 to 190 seconds.

The ACP packets encapsulate the DS op-codes inside an ACP over SDP packet and are transmitted using two techniques: *Serial* and *Interleaved*. In the *Serial* transmission we consecutively send all packets directed to a processor before changing the destination. Whereas, in the *Interleaved* transmission we change the destination for each packet so that we never send two consecutive packets to the same chip. We run both *Serial* and *Interleaved* transmission 20 times, one per each t_{pdd} value, in a 50 µs to 500 µs range using a 50 µs step.

During these runs, we counted how many APs completed the configuration and how many packets were lost. The loss of a packet is confirmed by the processor when the application receives a packet with an unexpected sequence number. On the Monitor Processor side, we counted the occurrences of *Software Error* (SWE). An SWE is the error triggered when the Root Processor fails to send an SDP to a target Monitor Processor. When too many SWEs occur, the Root Processor enters in the *Runtime Error (RTE)* status and it stops, making the SpiNNaker Board unreachable by the host. During our evaluation of the results, we considered the *(RTE)* a critical failure for the test. Conversely, we considered successful those tests terminated with SWE counter equal to zero.

The test environment consists of a host computer, equipped with an Intel Core i5-4670 @ 3.40GHz, 4 GB DDR3-RAM and running GNU/Linux Debian 8 (Jessie) distribution, a Gigabit Ethernet Switch and a SpiNN-5.

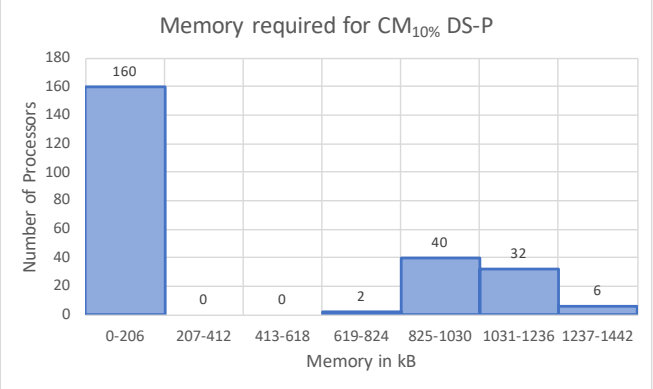

Figure 5: **DS-P Memory Footprint.** Per-processor memory usage of Data Specification Program leveraging SCP.

5.1.2 Serial transmission

In the first set of tests, we implemented a serial transmission for stressing the Application Processors and the destination Monitor Processor detecting the minimum t_{pdd} that guarantees a reliable transmission without any loss of packets. ACP over SDP packets are transmitted sequentially from the Root Node to the Application Processor until the configuration packages are consumed.

Table 1.SERIAL summarises the results of these stress tests. The table refers to all 20 simulations for each t_{pdd} chosen. We detected critical failure conditions (RTE $>$ 0) when t_{pdd} is in the range of 50-250 µs. In the case of t_{pdd} time equal to 50 µs, it was not possible to obtain any data because all the twenty simulations failed. This critical failure is due to a chain of events starting with the saturation of the shared message box between MP and AP that creates a deadlock condition overloading the MP in charge of dispatching the ACP packets. Considering t_{pdd} values above 250 µs the system can configure the 99.999% of the APs (reliability class of 9s equal to 5). From the results, we obtained a t_{pdd} of 300 µs to avoiding the saturation of buffers in Application Processors.

5.1.3 Interleaved transmission

We designed this benchmark to find the minimum threshold of t_{pdd} time that guarantees the correct transmission of all configuration packets from the Root Processor.

In this scenario we avoided the overloading of the target MP, stressing only the Root Processor. The host computer forwards configuration packets so that two consecutive packets are never sent to the same chip. For example, if we want to configure the AP-1 of the Chip-0-0, the AP-2 of the Chip-0-1 and the AP-3 of the Chip-1-0 respectively, the procedure works as follows:

1) ACP over SDP Packet-1-1 is sent to the MP of the Chip-0-0 and forwarded to the AP-1.
2) After a waiting time equal to t_{pdd}, the Packet-2-1 is sent from the Root Processor to the MP of the Chip-0-1 and forwarded to the AP-2.
3) The Packet-3-1 reaches the MP of the Chip-1-0 and is forwarded to the AP-3.
4) The Packet-1-2 is sent to the MP of the Chip-0-0 for the AP-1.

SERIAL

t_{pdd} [µs]	Configuration Packets		Root Processor		APs	
	Missed / Total	Class of 9s	SWE	RTE	FINISH	WAIT
50	**	**	**	100%	**	**
100	20 661 / 9 561 640	2	39 186	0%	63.9%	36.1%
150	1 570 / 9 561 640	3	28 273	0%	91.2%	8.8%
200	122 / 7 171 230	4	91	25%	96.7%	3.3%
250	71 / 9 083 558	5	96	5%	98.8%	1.2%
300	22 / 9 561 640	5	0	0%	99.5%	0.5%
350	13 / 9 561 640	5	0	0%	99.7%	0.3%
400	15 / 9 561 640	5	0	0%	99.6%	0.4%
450	9 / 9 561 640	6	0	0%	99.7%	0.3%
500	17 / 9 561 640	5	0	0%	99.6%	0.4%

INTERLEAVED

t_{pdd} [µs]	Configuration Packets		Root Processor		APs	
	Missed / Total	Class of 9s	SWE	RTE	FINISH	WAIT
50	136 089 / 956 164	0	131 070	90%	43.0%	57.0%
100	6 447 / 9 561 640	3	6 100	0%	57.1%	42.9%
150	314 / 9 561 640	4	179	0%	93.4%	6.6%
200	121 / 9 083 558	4	29	5%	97.3%	2.7%
250	19 / 9 561 640	5	1	0%	99.6%	0.4%
300	12 / 9 561 640	5	0	0%	99.7%	0.3%
350	26 / 9 561 640	5	0	0%	99.4%	0.6%
400	26 / 9 561 640	5	0	0%	99.4%	0.6%
450	9 / 9 561 640	6	0	0%	99.8%	0.2%
500	11 / 9 561 640	5	0	0%	99.8%	0.2%

Table 1: **Test results**. At variation of t_{pdd} these tables describe: i) The number of ACP packets lost and the relative class of 9s ($number\ of\ 9s\ in\ 1 - missed/total$). ii) The status of Root Monitor Processors in terms of Software error and Runtime exceptions. iii) The status of Application processors in term of percentage of them that receive all own packets.

This mechanism of interleaved transmission continues until the configuration is complete. In the final phase of the transmission, if the cores to be configured lie in a single chip the packet delivering delay is increased using a safety threshold of 1 ms, in order to avoid saturation of the target MP involved in the configuration of the last cores. Using this technique, we prevent the burst transmission of packets to the same SpiNNaker Chip, giving to the MP and to the APs a sufficient amount of time to reconstruct and process ACP packets.

The use of interleaved sending of configuration packets made it possible to analyse the limits of the Root Processor previously masked by the errors generated by the MPs of the target chips.

In Table 1.INTERLEAVED we reported the results of this test. We identified the critical failure condition (for all the 20 repetitions) that occurred when we imposed a delay time t_{pdd} of 50 µs and a single critical event for t_{pdd} equal to 200 µs. In the t_{pdd} range of 100 µs to 250 µs we detected several SWEs that indicate the lower bound imposed by the limits of the hardware component involved in the delivering process of packets inside the SpiNNaker Board. We identify the cause of this issue to a limitation of the Root Processor used to fragment the SDP packets into P2P packets. The system can configure the 99.999% of APs with a reliability class of 9s equal to 5 using values of t_{pdd} higher than 200 µs.

5.1.4 Profiling

The plots in Figure 6 show the occurrence of SWE when the t_{pdd} is increasing. We note a higher number of SWE when ACP over SDP packets are transmitted with the Serial method, thus validating our hypothesis that the interleaved transmission is a valid solution to the issues related to HW and time limits of target chips having to reconstruct and interpret incoming ACP packets. In the interleaved transmission the Root Node is responsible for the generation of SWEs.

We consider as successful transmissions those without errors. A transmission of this type can be detected for $t_{pdd} \geq 250\ \mu s$ for interleaved and $t_{pdd} \geq 300\ \mu s$ for serial loading. We highlighted, in the chart of Figure 6, the presence of RTE using outliers points. This condition happens at 200 µs for the Interleaved transmission and at 250 µs for the Serial transmission.

The missing packets counted at the target have a trend similar to SWE (see Figure 6). The number of missing packets reduce to a value near to 1.0E-06 for $t_{pdd} \geq 250\ \mu s$ in the Interleaved transmission and for $t_{pdd} \geq 300\ \mu s$ in the Serial transmission. Even in this situation, we can observe the same crash events represented as single points in the figure.

In summary, we identify an ideal delay time of 300 µs for the Serial transmission and of 250 µs for the Interleaved transmission.

In Figure 7 we can identify four t_{pdd} operating zones: i) Green area: no problems encountered, all packets are correctly transmitted; ii) Yellow area: only Interleaved transmission can terminate without error; iii) Red area: detected some issues, acceptable only when using Interleaved loading; iv) Below 150 µs (grey area) the system is unstable. The bandwidth profile shows the throughput for each operating zone: For yellow and green operating zones we reach values between 6 and 8 Mbit per second. Whereas, the red operating zone allows a bandwidth between 8 and 13 Mbit per second.

5.2 ACP for SNN Applications Reconfiguration

In this section, we describe a test designed to highlight the capability of the ACP when used for reconfiguring the application parameters at runtime during the simulation. For this test, we selected two different networks, where the neuron model was modified to support the features defined in the ACP.

The first network is a bio-inspired SNN for multivariate classification designed by Schmuker et al. [28]. This SNN is inspired by the chemical sense of insects evolved to encode and classify odorants in the natural environment. Schmuker et al. [28], based on these insights, developed a computational method to encode, process, and classify handwritten numbers.

The second network is an SNN composed of a chain of neurons that stimulate each other (Synfire Chain) [29]. The speed of propagation of neuronal stimuli (spikes) depends on the synaptic delay of the individual synapses that connect them.

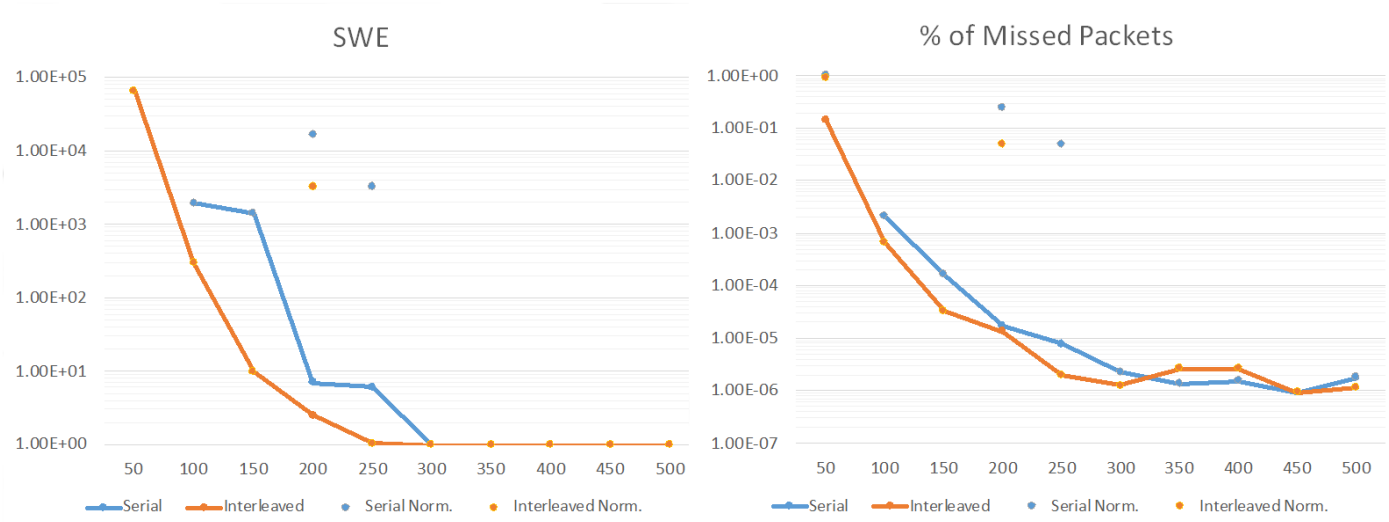

Figure 6: **Software Errors and percentage of missed packets for serial and interleaved transmission.** The plot on the left shows SWE at increasing values of t_{pdd} (µs). The lone points are values inserted to take RTE into account (the y-axis is in logarithmic scale, all values are incremented by 1 to avoid zeros). The plot on the right shows the number of missed packets at increasing values of t_{pdd} (µs). The lone points are values inserted to take RTE into account (the y-axis is in logarithmic scale and represents a percentage).

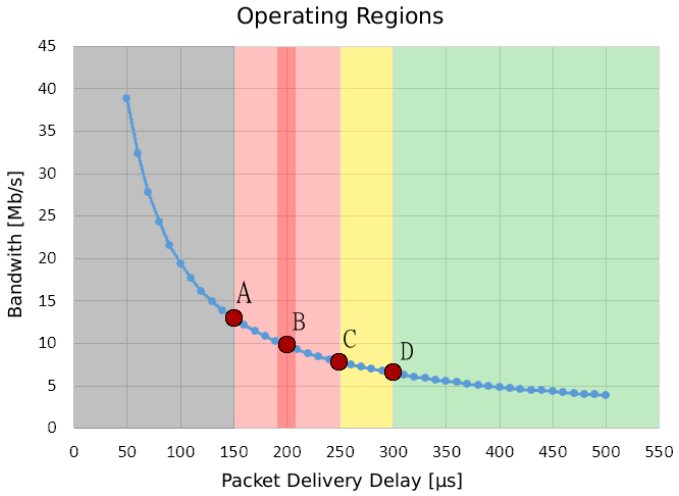

Figure 7: **Operating Region.** This plot depicts trend of communication bandwidth when t_{pdd} (µs) is increasing. The different operating zones are represented with different colours. Red points represent values of four different boundaries. A is the last recommended value with the interleaved sender: 13 Mbit/s. C is the last recommended value with the serial sender: 7.5 Mbit/s. D is a secure value that works for all senders: 6.5 Mbit/s. In B we get an unusual amount of RTE.

5.2.1 SNN Classifier

The SNN-Classifier has three functional layers. In the first layer, the original stimulus space is sampled by Virtual Receptors (VRs) which respond proportionally to the data input proximity, thus encoding the stimulus using cone-shaped radial basis functions with large overlapping receptive fields. The centroids of the basis functions (the VR points) were placed using the neural gas algorithm [30], a self-organising process to map the feature space described by a picture data set. In the second layer, the lateral inhibition decorrelated the signals from the VRs. Signals from Virtual Receptors (in the form of firing rates) reach the Receptor Neurons (RNs) modelled as Integrate and Fire (IF) neurons.

Each population of RNs excites a population of *Projection Neurons (PNs)*, which in turn send their spikes to one population of *Local Inhibitory neurons (LINs)*. Each LIN population sends inhibitory projections to all other PN populations in the second layer, exerting lateral inhibition, reducing the correlation between VR channels and scattering the representation of the multi-dimensional pattern. Signal decorrelation during the second layer significantly increases the classification accuracy. Finally, in the third layer, olfactory scent perception is modelled by a machine learning classifier able to classify the input data linearly.

The synapses with plasticity model are situated between the second and the third layer and are connected with a set of neurons that have the functionality of trainers since their signal selectively stimulates the synapses associated with the class to learn at a given instant.

The execution of the SNN classifier is divided into two execution phases: the training phase and the testing phase. During the training phase, the network is built with plastic synapses, and the trainer neurons are configured to emit a spike so that the submitted sample (the samples are presented for 200 ms) is coupled with the desired class. With this implementation, in the absence of the trainer neurons spikes, the plastic synapses deviate from the learned weights, causing the incorrect behaviour of the network. Hence, it is not possible to perform the test phase in the same simulation of the training.

Another network configuration is required to solve this issue. At the end of the Training phase, all the learned

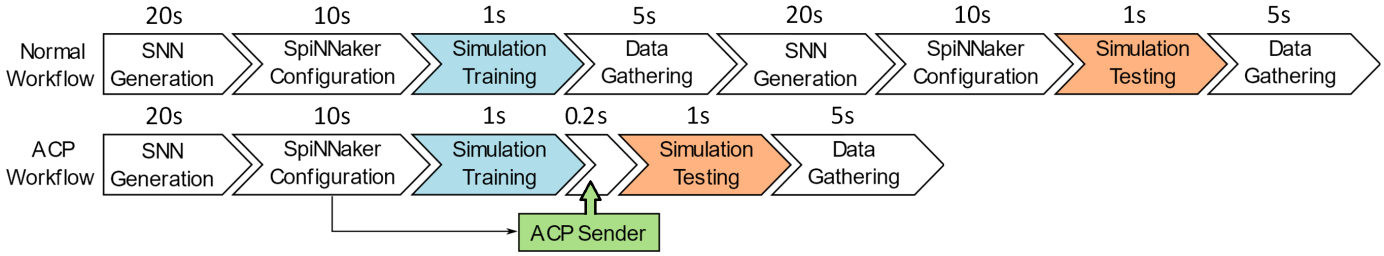

Figure 8: **Classifier Workflows.** Two different workflows to perform training and testing phases with the SNN classifier. Without the ACP the network uses the workflow to the top, the training and testing phases are two different simulations. With the ACP usage, the network uses the workflow to the bottom, the training and testing phases are in the same simulations.

weights are downloaded from the board and used to rebuild the classifier using static synapses neuron. By re-running this new network, it is possible to evaluate the classification performance of the network.

We show the workflow on the top of Figure 8, where all the operations to simulate an SNN classifier, require about 72 s. This time is the overall period required to perform two generation phases (20 s each), two board configuration phases (10 s each), two simulation phases (1 s each), a download phase for recovering trained synapses weights (5s), and a download phase for collecting classification results (5s).

We implemented the ACP inside the neuron application to improve the overall simulation time. The application uses an ad-hoc command to change the synaptic plasticity behaviour. On the host side, an application (the Sender) making use of the SpynnakerACP library, is in charge of sending all processors the command for disabling the learning capabilities of plastic synapses. The Sender is executed when the simulation is ready to be run on the board.

In this way, it is possible to perform both Training and Testing phases with a single simulation and the usage of the ACP Sender application configured to inhibit the plastic synapses after 2 s from the start of the simulation. By doing so, both the phases of the classifier are performed in just about 38 s (2.2 s of simulation). We added 200 ms between learning and testing phases to give time to the ACP Sender to propagate packets to all the cores. This workflow is depicted in Figure 8.

ACP Sender transmits with a Packet Delivery Delay time, t_{pdd}, set to 200 µs, allowing a safe transmission of the switching packet to all 864 cores of a SpiNNaker Board in about 170 ms. During the tests, we stressed the system running 12 instances of classification network at the same time. As a result, we correctly sent all the ACPoverSDP packets to all 864 cores involved in the simulation. The router overload resulted in some missed MC packets, representing the spikes during the simulation, but this did not impact on the performance of the network and the classification results.

The significant advantage provided by the ACP embedded in the classifier SNN is the possibility to run both training and test phases without reconfiguring the board.

5.2.2 SNN Synfire Chain

The second application based on SNN is the Synfire Chain. The network is composed of a long sequence of neurons linked together by a single synapse. Currently, the simulation once configured and started does not allow to modify any parameter. Unless making usage of synapse models whose weights vary autonomously, the only way to modify the parameters of the neuronal model such as weight and synaptic delay is to know the position in RAM of the data regions related to each neuron and through SCP commands directly modify the content of the memory.

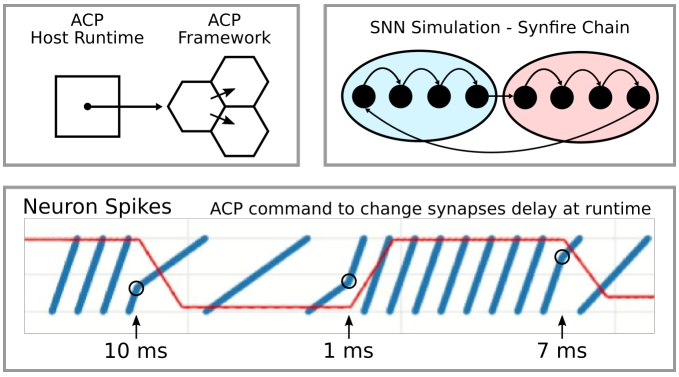

Figure 9: **ACP reconfiguration of neuronal parameters.** Above: the test configuration and the benchmark network used. Below: the graph of the spikes emitted by the neurons (blue), mean network activity (red). The series of spikes changes slope after receiving the synaptic delays reconfiguration commands at 10 ms, 1 ms and 7 ms.

We used the memory entities provided by the ACP framework to manage a set of parameters as modifiers of the actual model parameters, without modifying the whole configuration plan of the neural application. By doing so, we were able to modify the delay of the synapses in real-time, during the simulation of the synfire chain. Figure 9 depicts a scheme of SNN, the ACP components involved and the timeline of an SNN simulation. The user can dynamically change the corresponding parameters (e.g. synapse delay) using the ACP runtime library to send commands to the SpiNNaker cores to modify the memory content exploiting Memory Entities support.

Figure 9 shows the trend of the network spike series (blue lines) and the average synaptic activity of the whole network (red line). Circles are used to highlight the points at which the commands are triggered to modify the memory

entities, which affects the synaptic delay between the neurons. The slope of the spike series increases with a minimum delay (1 ms) and decreases when the synaptic delay is set to a higher level (10 and 7 ms).

This case study demonstrates that ACP framework, by allowing runtime reconfigurations, can be used for effective host-controlled SNN parameters exploration.

6 Conclusions

We designed the Application Command Protocol, a new method to be adopted at the application level for spreading commands and manage the memory of the SpiNNaker neuromorphic platform. ACP allows the users to include in their distributed applications the subset of commands to carry out only the needed activities, hence saving memory for the code. On top of that, ACP allows the exchange of commands between application processors without involving the respective monitor processors using the multicast channel, thus optimising the communication flow. It provides a useful abstraction level of the memory which users can easily access through a virtual id to all the variables of the applications running on one application processors (AP) from any other AP of the system.

We modify two SpiNNaker applications in order to use the ACP. We inserted two different ACP implementation: one inside the application used during the *Configuration* of SpiNNaker board, and another one embedded into a neuron model used during the SNN *Simulation* phase. The ACP implementation in the first application uses a heavy interpreter and enables a flexible and optimised set-up of the board during the configuration phase. Conversely, the ACP implementation in the second application is lighter and can be easily embedded into the neuron model applications, allowing the user to change some parameters a run-time during the simulation phase.

In the first application, we demonstrated the advantages introduced by ACP interpreter in the run-time feeding of configuration applications. More specifically, we analysed the behaviour of the Monitor Processor of the node attached to the Ethernet interface, that is in charge of managing the communication with the external sources. Handling the configuration phase at the application level with ACP allows the configuration to be performed by all kinds of external sources. The use of this new method is straightforward and can speed-up host-to-boards data transmission during the configuration of SpiNNaker platforms.

In the second application, we demonstrated the run-time flexibility introduced by the ACP interpreter embedded in the neuron model application, implementing two different real simulation scenarios: i) a two-phases SNN-Classifier designed for discriminating the handwritten number and ii) a chain of neurons with run-time re-configuration parameters. Our results show that ACP allows to switch between training and testing phase in half of the time needed by the former workflow and to change model parameters (e.g. synapse delay) during the simulation.

Both the implemented applications were demonstrated to be flexible, scalable and expandable.

In the end, we believe that this work opens the way to more flexible use of many-core neuromorphic platforms as brain simulators and as support for new computational brain-inspired paradigms. The same concepts leveraged by ACP can be exported to different types of platforms (e.g. Intel Loihi), addressing a much broader group of problems.

Acknowledgments

The research leading to these results has received funding from European Union Horizon 2020 Programme [H2020/2014-20] under grant agreement no.720270 [HBP-SGA1]. We also thank the APT Group of Manchester University for providing us the SpiNNaker Board, and in particular S.Furber, A.Rowley, A.Stokes and D.Lester for their constructive suggestions.

References

[1] P. Merolla, J. Arthur, F. Akopyan, N. Imam, R. Manohar, and D. S. Modha, "A digital neurosynaptic core using embedded crossbar memory with 45pj per spike in 45nm," in *2011 IEEE Custom Integrated Circuits Conference (CICC)*, Sept 2011, pp. 1–4.

[2] K. Meier, "A mixed-signal universal neuromorphic computing system," in *2015 IEEE International Electron Devices Meeting (IEDM)*, Dec 2015, pp. 4.6.1–4.6.4.

[3] G. Indiveri, E. Chicca, and R. Douglas, "A vlsi array of low-power spiking neurons and bistable synapses with spike-timing dependent plasticity," *IEEE transactions on neural networks*, vol. 17, no. 1, pp. 211–221, 2006.

[4] S. Furber, D. Lester, L. Plana, J. Garside, E. Painkras, S. Temple, and A. Brown, "Overview of the spinnaker system architecture," *Computers, IEEE Transactions on*, vol. 62, no. 12, pp. 2454–2467, Dec 2013.

[5] W. Maass, "Networks of spiking neurons: the third generation of neural network models," *Neural networks*, vol. 10, no. 9, pp. 1659–1671, 1997.

[6] M. Davies, N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain *et al.*, "Loihi: A neuromorphic manycore processor with on-chip learning," *IEEE Micro*, vol. 38, no. 1, pp. 82–99, 2018.

[7] H. Paugam-Moisy and S. Bohte, "Computing with spiking neuron networks," in *Handbook of natural computing*. Springer, 2012, pp. 335–376.

[8] C.-K. Lin, A. Wild, G. N. Chinya, Y. Cao, M. Davies, D. M. Lavery, and H. Wang, "Programming spiking neural networks on intel's loihi," *Computer*, vol. 51, no. 3, pp. 52–61, 2018.

[9] O. Rhodes, P. A. Bogdan, C. Brenninkmeijer, S. Davidson, D. Fellows, A. Gait, D. R. Lester, M. Mikaitis, L. A. Plana, A. G. Rowley *et al.*, "spynnaker: A software package for running pynn simulations on spinnaker," *Frontiers in neuroscience*, vol. 12, 2018.

[10] X. Jin, M. Lujan, L. Plana, S. Davies, S. Temple, and S. Furber, "Modeling spiking neural networks on spinnaker," *Computing in Science Engineering*, vol. 12, no. 5, pp. 91–97, Sept 2010.

[11] A. D. Rast, X. Jin, F. Galluppi, L. A. Plana, C. Patterson, and S. Furber, "Scalable event-driven native parallel processing: The spinnaker neuromimetic system," in *Proceedings of the 7th ACM International Conference on Computing Frontiers*, ser. CF '10. New York, NY, USA: ACM, 2010, pp. 21–30. [Online]. Available: http://doi.acm.org/10.1145/1787275.1787279

[12] A. D. Brown, J. S. R. Steve B. Furber, L. A. P. Jim D. Garside, Kier J. Dugan, and S. Temple, "Spinnaker programming model," *IEEE Transactions on Computers*, vol. 64, no. 6, pp. 1769–1782, June 2015.

[13] A. Rast, F. Galluppi, S. Davies, L. A. Plana, T. Sharp, and S. Furber, "An event-driven model for the spinnaker virtual synaptic channel," in *Neural Networks (IJCNN), The 2011 International Joint Conference on*, July 2011, pp. 1967–1974.

[14] A. D. Rast, J. Partzsch, C. Mayr, J. Schemmel, S. Hartmann, L. A. Plana, S. Temple, D. R. Lester, R. Schuffny, and S. Furber, "A location-independent direct link neuromorphic interface," in *Neural Networks (IJCNN), The 2013 International Joint Conference on*. IEEE, 2013, pp. 1–8.

[15] G. Urgese, F. Barchi, and E. Macii, "Top-down profiling of application specific many-core neuromorphic platforms," in *IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-15) (IEEE MCSoC-15)*, Turin, Italy, sep 2015.

[16] G. Urgese, F. Barchi, E. Macii, and A. Acquaviva, "Optimizing network traffic for spiking neural network simulations on densely interconnected many-core neuromorphic platforms," *IEEE Transactions on Emerging Topics in Computing*, vol. pp, no. 99, 2016.

[17] F. Barchi, G. Urgese, E. Macii, and A. Acquaviva, "Impact of graph partitioning on snn placement for a multi-core neuromorphic architecture: Work-in-progress," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 4:1–4:2. [Online]. Available: http://dl.acm.org/citation.cfm?id=3283552.3283556

[18] A. P. Davison, D. Brüderle, J. Eppler, J. Kremkow, E. Muller, D. Pecevski, L. Perrinet, and P. Yger, "Pynn: a common interface for neuronal network simulators," *Frontiers in neuroinformatics*, vol. 2, 2008.

[19] T. U. o. Manchester, "Pacman," 2015. [Online]. Available: https://github.com/SpiNNakerManchester/PACMAN

[20] F. Barchi, G. Urgese, A. Acquaviva, and E. Macii, "Directed graph placement for snn simulation into a multi-core gals architecture," in *Very Large Scale Integration (VLSI-SoC), 2018 IFIP/IEEE International Conference on*. IEEE, 2018, p. 4.

[21] J. Navaridas, M. Luján, L. A. Plana, S. Temple, and S. B. Furber, "Spinnaker: Enhanced multicast routing," *Parallel Computing*, vol. 45, pp. 49–66, 2015.

[22] G. Liu, P. Camilleri, S. Furber, and J. Garside, "Network traffic exploration on a many-core computing platform: Spinnaker real-time traffic visualiser," in *Ph.D. Research in Microelectronics and Electronics (PRIME), 2015 11th Conference on*, June 2015, pp. 228–231.

[23] S. Temple, "Appnote 4 - spinnaker datagram protocol (sdp) specification," 2011, available at https://spinnaker.cs.manchester.ac.uk/.

[24] ——, "Appnote 5 - spinnaker command protocol (scp) specification," 2011, available at https://spinnaker.cs.manchester.ac.uk/.

[25] F. Barchi, G. Urgese, E. Macii, and A. Acquaviva, "An efficient mpi implementation for multi-core neuromorphic platforms," in *Proceedings of the 1st IEEE conference on New Generation of Circuits and Systems Conference*. IEEE, 2017, p. 4.

[26] A. Siino, F. Barchi, S. Davies, G. Urgese, and A. Acquaviva, "Data and commands communication protocol for neuromorphic platform configuration," in *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSOC)*, Sept 2016, pp. 23–30.

[27] T. C. Potjans and M. Diesmann, "The cell-type specific cortical microcircuit: relating structure and activity in a full-scale spiking network model," *Cerebral Cortex*, vol. 24, no. 3, pp. 785–806, 2014.

[28] M. Schmuker, T. Pfeil, and M. P. Nawrot, "A neuromorphic network for generic multivariate data classification," *Proceedings of the National Academy of Sciences*, vol. 111, no. 6, pp. 2081–2086, 2014.

[29] M. Abeles, "Synfire chains," *Scholarpedia*, vol. 4, no. 7, p. 1441, 2009, revision #150018.

[30] T. Martinetz, K. Schulten *et al.*, *A" neural-gas" network learns topologies*, 1991.

Francesco Barchi is PhD Student at the Dept. of Control and Computer Engineering of Politecnico di Torino. He received his M.Sc. degree in Computer Engineering at Politecnico di Torino. He has experience in multidisciplinary task with a collaboration with GAMUT s.r.l for the development of a MASW analysis software. He designed, during his M.Sc. thesis, an optimized technique for partitioning and placement of SNN in the SpiNNaker neuromorphic platform. His research interests focus on optimization problems and software develop for bioinformatics algorithms: MD, SNN and sequence alignment on heterogeneous platform, from GPU to multi-core architectures.

Gianvito Urgese is Postdoc at the Dept. of Control and Computer Engineering of Politecnico di Torino. He received his M.Sc. degree (summa cum laude) in Electrical Engineering at Politecnico di Torino. He designed, during his M.Sc. thesis, an optimized HW accelerator for sequence alignment, implemented in VHDL on a systolic array architecture. He was, in 2011, research trainee at Teseo S.p.A. involved in the design of a system for structural health monitoring in composite materials. In 2008 he collaborate with the INRIM institute for a project concerning the redefinition of Boltzmann constant. His research interests focus on: (i) Research and design of optimized task-specific bioinformatics algorithms; (ii) Development of tools for the study of non coding biological sequences (miRNA, siRNA and lncRNA); (iii) Design of heterogeneous SW/HW architectures to accelerate bioinformatics algorithms, including parallel implementation on FPGA and GPU; (iv) Design of partitioning and placement algorithms to map Spiking Neural Networks in the SpiNNaker neuromorphic platform.

Alessandro Siino is a Computeer Engineer, he received his M.Sc. degree in Computer Engineering at Politecnico di Torino. He designed and implemented, during his M.Sc. thesis, a new protocol for SpiNNaker architecture configuration, in that period he was Visiting Academic at The University of Manchester where had the opportunity to acquire deep knowledge of brain inspired architectures and SoC configuration. His interests focus on embedded system, software development and in the field of enterprise systems integration.

Santa Di Cataldo received her Biomedical Engineering degree (summa cum laude) from Politecnico di Torino, Italy, in 2006. In 2007 she joined the Department of Control and Computer Engineering (DAUIN) in Politecnico di Torino as a Research Assistant. She holds a Ph.D. in Systems and Computer Engineering from the same university since April 2011. She is currently Assistant Professor. Her main research interests are biomedical image processing and data mining, including techniques for pattern recognition, image segmentation, quantification and classification of image features for medical and biological applications. She has experience in the design and development of image analysis tools in several imaging modalities, including B- mode Ultrasound, high-resolution microscopy and Magnetic Resonance Imaging.

Enrico Macii is a Full Professor of Computer Engineering at Politecnico di Torino. From 1991 to 1997 he was also an Adjunct Faculty at the University of Colorado at Boulder. He holds a PhD degree in Computer Engineering from Politecnico di Torino (1995). Since 2007, he is the Vice Rector for Research and Technology Transfer at Politecnico di Torino, and since 2012 also the Rector's Delegate for International Affairs. He was the National FP7 ICT Delegate from 2011 until 2013, and one of the Italian Members of the Public Authorities Board of the ENIAC and ARTEMIS Joint Undertakings from 2009 until 2013. His research interests are in the design of electronic digital circuits and systems, with particular emphasis on lowpower consumption aspects. In the field above he has authored around 450 scientific publications.

Andrea Acquaviva is Associate Professor at Politecnico di Torino, Italy. He received the Ph.D. degree in electrical engineering from the University of Bologna, Italy, in 2003. In 2003, he became an Assistant Professor with the Computer Science Department, University of Urbino, Italy. From 2005 to 2007, he was a Visiting Researcher with the Ecole Polytechnique Federale de Lausanne, Switzerland. In 2006, he joined the Department of Computer Science, University of Verona, Italy. He has been with the Department of Computer Engineering and Automation, Politecnico di Torino. His research interests focus mainly on parallel computing for distributed embedded systems such as multi-core and sensor networks and simulation and analysis of biological systems using parallel architectures. In the fields above, he has authored over 140 scientific publications.