spiNNlink: FPGA-Based Interconnect for the Million-Core SpiNNaker System

DOI:
10.1109/ACCESS.2020.2991038

Document Version
Final published version

Link to publication record in Manchester Research Explorer

Citation for published version (APA):

Published in:
IEEE Access

Citing this paper
Please note that where the full-text provided on Manchester Research Explorer is the Author Accepted Manuscript or Proof version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version.

General rights
Copyright and moral rights for the publications made accessible in the Research Explorer are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Takedown policy
If you believe that this document breaches copyright please refer to the University of Manchester’s Takedown Procedures [http://man.ac.uk/04Y6Bo] or contact uml.scholarlycommunications@manchester.ac.uk providing relevant details, so we can investigate your claim.
spiNNlink: FPGA-Based Interconnect for the Million-Core SpiNNaker System

LUIS A. PLANA¹, JIM GARSIDE², JONATHAN HEATHCOTE², JEFFREY PEPPER¹, STEVE TEMPLE³, SIMON DAVIDSON¹, MIKEL LUJÁN¹, AND STEVE FURBER¹, (Fellow, IEEE)

¹Department of Computer Science, The University of Manchester, Manchester M13 9PL, U.K.
²BBC Research and Development, Salford M50 2LH, U.K.
³Mindtrace, Manchester M3 3BQ, U.K.

Corresponding author: Luis A. Plana (luis.plana@manchester.ac.uk)

This work was supported in part by EPSRC under Grant EP/D07908X/1 and Grant EP/G015740/1, in part by the European Research Council under grant agreement ERC/(FP7/2007-2013)/320689, and in part by the EU Human Brain Project under Grant FP7-604102, Grant H2020 720270, and Grant H2020 785907. The work of Luis A. Plana was supported by the RAIN Hub, which is funded by the Industrial Strategy Challenge Fund, part of the government’s modern Industrial Strategy. The fund was delivered by UK Research and Innovation and managed by EPSRC under grant EP/R026084/1.

ABSTRACT SpiNNaker is a massively-parallel computer system optimized for the simulation, in real-time, of very large networks of spiking neurons. The system consists of over 1 million, energy-efficient ARM cores distributed over 57,600 SpiNNaker chips, each of which contains 18 cores interconnected by a neurobiologically-inspired, asynchronous (clock-less) Network-on-Chip. The NoC is extended to the chip boundary for chip-to-chip communication. To construct the massively-parallel system, SpiNNaker boards, housing 48 SpiNNaker chips, are connected together using FPGA-based, high-speed serial links. This paper presents some of the novel aspects of the design and implementation of the bespoke interconnect, including a credit-based, reliable frame transport protocol that allows the multiplexing of asynchronous SpiNNaker channels over the serial links, and an efficient FPGA-to-SpiNNaker chip interface that provides twice the throughput of traditional asynchronous interfaces. SpiNNaker houses 3,600 Xilinx Spartan-6 FPGAs, provides a bisection bandwidth of 480 Gbit/s, and ran the first-ever, true real-time brain cortical simulation [1] – a feat not currently achievable using conventional HPCs or GPUs.

INDEX TERMS High-speed interconnect, field-programmable gate array (FPGA), asynchronous interface, neuromorphic or neurobiologically-inspired computing.

I. INTRODUCTION Understanding how the brain works is one of the outstanding scientific grand challenges. SpiNNaker [2], the Spiking Neural Net Architecture, is a massively-parallel multi-core system optimized for the simulation, in real time, of very large networks of spiking neurons. The million-core system, shown in Fig. 1, was designed as an energy-efficient platform to support research into how the brain processes information and into its fault-tolerant capabilities.

The brain is a very active area of research. Although there is an important body of knowledge about how the different components of the brain - such as neurons, synapses and dendrites - operate, it is not clear what are the best ways to model how data is processed in the brain. On the other hand, it is well established that neurons communicate using spikes, i.e., short stereotypical pulses that convey only two pieces of information: which neuron sent the spike and the time at which it was sent. A neuron typically multicasts every spike to many thousands of other neurons.

Simulating neural nets that have realistic numbers of neurons and synapses and plausible levels of spiking activity is extremely challenging and requires careful evaluation. The cortical microcircuit model [1] is a well-established benchmark that illustrates the current scale of simulations. The model represents approximately 1 mm² of brain cortex, containing 80,000 neurons and 0.3 billion synapses.

To maximize flexibility as a research platform, SpiNNaker uses software to simulate neural devices. This allows researchers to represent models in any way they want...
and to experiment with them. They can even introduce new devices that so far have not been considered in the simulations. However, there is no system-wide shared memory. This enforces the use of a message-passing communication mechanism, similar to the one used in the brain.

As opposed to many massively-parallel systems that use interconnects such as InfiniBand and Ethernet [3] to optimize point-to-point transfers of large blocks of data through MPI [4] or similar mechanisms, the neurobiologically-inspired SpiNNaker communications infrastructure has been designed to carry efficiently short messages (spikes) concurrently to many different destinations.

The SpiNNaker asynchronous—truly clock-less—interconnect allows a seamless integration of the communication infrastructure at different levels. The bespoke on-chip network extends to the inter-chip interconnect and, as the size of the system grows, the chip-to-chip interconnect extends into board-to-board links. These inter-board links, called spiNNlinks, are the subject of the work described here.

The rest of this manuscript is organized as follows: Section II provides a short introduction to the SpiNNaker architecture and its interconnect. The implementation of spiNNlink, the board-to-board interconnect, is presented in Section III. Section IV covers a novel synchronization scheme used in spiNNlink to interface with the asynchronous SpiNNaker channels. Finally, an assessment of spiNNlink and our conclusions are presented in Sections V and VI.

II. SpiNNaker

The SpiNNaker architecture [2] is based around the multi-die SpiNNaker chip [5]. The 130 nm feature-size SpiNNaker die houses 18 energy-efficient ARM968 cores, measures 10 mm² and contains approximately 100 million transistors, mostly as static RAM. The chip also houses a 1 Gbit mobile DDR SDRAM die, wire-bonded to the SpiNNaker die.

The SpiNNaker chip is organized as a Globally-Asynchronous, Locally-Synchronous (GALS) system, i.e., the 18 synchronous core subsystems are interconnected by an energy-efficient, clock-less, packet-switching Network-on-Chip (NoC) [6]. The NoC fabric uses delay-insensitive codes [7] and handshakes to guarantee correct packet delivery without the use of clocks. The cores are organized in a star topology around the synchronous SpiNNaker router. A sensible performance–energy balance is reached when the core subsystems are run at 200 MHz, the SDRAM interface is set to 130 MHz and the SpiNNaker router is run at 133 MHz, as it usually meets its traffic demands at this speed. When all 18 cores operate at full load the chip dissipates around 1 W.

A. SpiNNaker PACKETS

As their main function is to represent spikes, SpiNNaker packets are very short, either 40 or 72 bits. A packet consists of an 8-bit control header, a 32-bit routing key (used by the SpiNNaker router to determine the packet destinations), and an optional 32-bit payload (not normally used in packets representing spikes).

Without going into the details of the routing mechanism [2], [5], [6], it is interesting to note that the SpiNNaker router can multicast packets from one core to many destinations without any software intervention. The key insight for efficient multicasting is that the routing key identifies the source of the packet, not its destination. The destinations are stored in a distributed way in configurable routing tables present in every router. Packet destinations can be any subset of the 18 on-chip cores in addition to any subset of the neighboring chips, accessed through the chip-to-chip channels described in the following section.

B. SpiNNaker CHIP-TO-CHIP INTERCONNECT

The energy-efficient NoC encoding is ideal for inter-chip interconnect as it allows each channel to operate at its own speed and is guaranteed to work correctly in the presence of unbounded interconnect delays. On SpiNNaker chip-to-chip channels, data is transmitted using the delay-insensitive 2-of-7 code [8]. In this code, the transmitter uses 7 wires to represent a 4-bit data value, and the receiver uses one wire as an acknowledgment to complete a handshake.

Fig. 2 shows how the code works: A 4-bit value is represented by exactly 2 wires changing value (either 0→1 or 1→0). These transitions can happen at any time and do not need to be simultaneous. A new value can be sent only after the transmitter has received an acknowledgement from the receiver (indicated by a change in value of the acknowledge wire). As the channels are bidirectional, each one requires 16 wires in total. Fig. 3 shows a histogram of the 4,608 chip-to-chip channel throughputs in one of the fifty 24-board card frames that make up SpiNNaker, clearly illustrating the clock-less nature of the channels (synchronous channels would all run at the speed of the clock and provide a homogeneous throughput).

C. SpiNNaker BOARD ORGANIZATION

Fig. 4 is a photograph of a SpiNN-5 board, used to build machines that can be as small as 1 board and as large as...
FIGURE 2. Operation of the 2-of-7 delay-insensitive code.

FIGURE 3. SpiNNaker chip-to-chip asynchronous channel throughputs.

FIGURE 4. SpiNNaker SpiNN-5 board.

FIGURE 5. SpiNNaker board 48-chip hexagon with FPGA connections.

1,200 boards. The board houses 48 SpiNNaker chips that contain 864 ARM cores in total. SpiNNaker systems use a two-dimensional hexagonal topology, illustrated in Fig. 5, in which each SpiNNaker chip is connected to six neighbors through the SpiNNaker channels described above.

In the photograph, the 48 SpiNNaker chips appear to form a square but they are actually organized as a hexagon, as shown in Fig. 5. This shape allows efficient logical board tiling to create multi-board machines [9]. The hexagonal organization of the chips is also efficient in the number of SpiNNaker channels on the board boundary. This is extremely important, given that, as shown in Fig. 5, 48 SpiNNaker channels are located at the board periphery and need to be connected to neighboring boards. If these channels were used directly to connect chips on different boards, an impractical total of 768 wires would be required. To reduce the number of wires, the SpiNNaker channels are multiplexed over High-Speed Serial Links (HSSLs) provided by three FPGAs and six HSSL connectors highlighted in Fig. 4. Each FPGA handles the 16 SpiNNaker channels on two adjacent sides of the hexagon, as shown in Fig. 5, and has spare capacity to manage peripheral connections, such as the neuromorphic channels described in [10]. The following section provides details of the board-to-board interconnect.

III. spiNNlink INTERCONNECT

Fig. 6 shows schematically how the inter-board HSSLs are used to multiplex SpiNNaker channels. The figure shows two neighboring boards, labeled A and B, connected through a HSSL. Each HSSL is used to multiplex eight SpiNNaker channels (although, for clarity, only two are shown in the figure). A data stream, labeled (d) in the figure, flowing from board A to board B, and the corresponding control stream, (c), flowing from board B to board A, will be described in the following section. Equivalent streams, not shown, flow in opposite directions for communications from board B to board A. Some of the novel aspects of the spiNNlink implementation address the challenge of using efficiently the HSSL bandwidth when multiplexing through it multiple lower-throughput, asynchronous channels.

A. spiNNlink OPERATION

Transmission over the HSSL is structured in frames. The different frame formats are shown in Fig. 7. There is one data frame type (data) and four frame types associated with
transmission control: out-of-credit (ooc), acknowledge (ack), reject or negative acknowledge (nack), and channel flow control (cfc). Each frame is identified by a start-of-frame special character (highlighted in red in the figure), contains a frame color (fc) and is protected by a CRC checksum (CRC). Frame types data, ooc, ack and nack also carry a sequence number (sequence), used to guarantee that frames are received in order. Two additional frame types, clock correction (clkc), and idle (idle), are used to keep the HSSL synchronized. To optimize bandwidth use, data frames also carry up-to-date ack and cfc fields, highlighted in green in the figure, that correspond to the opposite-direction control stream, allowing 100% bandwidth utilization by data frames on error-free streams.

Control frames are a single 32-bit word long while data frames have a variable length. As indicated earlier, eight SpiNNaker channels are multiplexed into a single HSSL and, for that purpose, a data frame can carry up to eight SpiNNaker packets, one from each channel. The 8-bit presence field is a bitmap used to indicate if the frame carries a packet from the respective channel. Similarly, the 8-bit length field is a bitmap that indicates if the packet is long (contains a payload) or short (no payload). This two bitmaps, part of the first word of every data frame, establish the actual structure and length of the frame. Depending on the number of SpiNNaker packets carried, data frames can be 4 to 20 32-bit words long. An alternative, non-packet-based data frame structure was explored in an early HSSL design [11] but no mechanism to deal with the difference in asynchronous channel bandwidths or idle channels was reported.

To send data across the HSSL, the transmitter (Tx) collects a packet from each SpiNNaker channel that has one available, assembles a data frame and sends it across. Tx stores the transmitted data in case retransmission is required. If at any point there is no data to send, idle frames are transmitted to keep the HSSL stream flowing and the receiver (Rx) synchronized. Multiple data frames can be assembled and sent successively, subject to a credit limit. If the Tx exhausts its credit, it sends ooc frames to request additional credit. Rx accepts frames and processes them according to their type. data, ooc and clkc frames are processes by Rx itself, and cfc, ack and nack frames are delivered to TX. idle frames are discarded after recording its 16-bit data value in a register. This value can be used for diverse purposes including, for example, the verification that cables have been connected correctly during machine assembly.

Received data frames are checked for framing or CRC errors. A correct data frame will have the expected sequence number, the correct color and pass its error checks. Correct frames are disassembled and the packets are sent to the corresponding SpiNNaker channels. These frames are acknowledged back to Tx, either using an ack frame or including the ack in a data frame flowing in the opposite direction, allowing it to release the stored data and providing additional credit. Erroneous frames, including an unexpected sequence number, generate a negative acknowledge, nack, which triggers a retransmission and also changes Rx color so that subsequent
frames can be flushed until the retransmitted one, in the right color, is received.

Rx provides updates on its status, *ack* and *cfc*, to Tx at expedient intervals. These are not necessarily triggered by data arrival and continue in the absence of new data. Information is conveyed on the channel flow control, credit available and current color. Error tolerance is provided by the repetition of these frames.

Tx updates its state on reception of Rx status. It re-credits and the sent data retained for retransmission is discarded up to any acknowledged sequence number. When an error indication, *nack*, is received Tx changes color and ignores further prompts until the data stream is re-established. It also resets its inputs to the *nack* sequence number and retransmits frames from that point. There is no requirement that the data contained is the same as in the original frames as frames may be reformed with additional data if available.

### B. *spiNNlink* IMPLEMENTATION

As mentioned earlier, *spiNNlink* can multiplex eight SpiNNaker channels across a HSSL. Fig. 8 shows a *spiNNlink* top-level block diagram, although, for clarity, only two incoming and two outgoing SpiNNaker channels are shown. The figure shows the main *spiNNlink* blocks: a link receiver (*link_rcv*) and a link sender (*link_snd*) for each SpiNNaker channel, the channel multiplexer/demultiplexer (*mux*), the transmitter and receiver controllers (*txc* and *rxc*), and the GTP gigabit transceiver [12], which is IP provided on the FPGA. Finally, a register bank, accessed through an SPI interface, is used to configure *spiNNlink* and to read its state. The link receivers and senders are clocked at a higher frequency than the rest of the modules and, therefore, synchronization FIFOs (*async_FIFO*), are required for data crossing between these modules and the multiplexer.

As detailed in [12], the GTP transceiver implements the HSSL physical layer through the Physical Media Attachment (PMA) and the Physical Coding Sublayer (PCS). The PMA contains the serializer/deserializer (SERDES) and the clock recovery circuitry. The PCS contains the 8b/10b encoder, the COMMA alignment and the elastic RX buffer with clock correction support. The elastic buffer duplicates or discards *clkc* frames to avoid becoming empty or full and losing synchronization. The transmitter and receiver controllers interface with the GTP and provide (TxC) and consume (RxC) data at an appropriate speed to keep the HSSL running. They also provide and consume clock correction frames at the correct rate for worst case drift. Finally, TxC scrambles data to reduce EMI. HSSL parameters such as voltage swing, TX pre-emphasis and RX equalization can be configured through the register bank.

The top-level implementation, shown in Fig. 8, is structured as a pipeline to optimize throughput. The modules share a common interface consisting of data (*data*), valid (*vld*) and ready (*rdy*) signals. If *vld* is asserted then data is ready for sending. The *rdy* signal is asserted when data can be accepted. Data is transferred between modules when both signals are asserted in the same clock cycle. The pipelined implementation of the modules allows data to be transferred in consecutive clock cycles, as long as both *vld* and *rdy* remain asserted.

The multiplexer is the central *spiNNlink* component. On the transmitter side it multiplexes incoming packets from eight SpiNNaker channels and assembles the HSSL frames while on the receiver side it disassembles the HSSL frames and demultiplexes the packets onto eight outgoing SpiNNaker channels. Fig. 9 shows the multiplexer block diagram (as in previous figures, for clarity only two incoming and two outgoing SpiNNaker channels are shown), which consists of four major modules: a frame assembler (FA) and a frame transmitter (FT) on the transmitter side, and a frame disassembler (FD) and a packet dispatcher (PD) on the receiver side. A register bank tracks relevant data such as transmitted and received frame counts, errors, channel flow status and credit.

---

**FIGURE 8.** *spiNNlink* top-level block diagram.

**FIGURE 9.** *spiNNlink* multiplexer block diagram.

FA contains a packet store (PS) for every incoming channel which, as the name implies, is used to store packets received...
from the channels to be sent across the HSSL. FA also contains a frame issue module that assembles data frames, as described earlier, and requests their transmission by FT. The frame issue module does not arbitrate or prioritize channels given that a data frame can accommodate one packet from every channel, adjusting the size of the frame according to the actual number of packets present. The packets are kept in the PS even after they have been sent in a frame, in case there is a need for retransmission. Section III-C below describes some of the novel aspects of the PS.

FA receives remote credit, channel flow control and ack/nack information via FD in order to assemble frames correctly. Details of the use of credit and flow control are given in Section III-D below. FT multiplexes the data and control streams that flow through the HSSL and computes a CRC checksum that is attached to every frame.

On reception of a frame, FD verifies the CRC checksum and the correct frame structure, signaling errors if any are detected. Received control information is directed to FA while data is passed to PD for delivery to the outgoing channels. PD contains a packet FIFO for each channel and stores the received packets until they are dispatched to the SpiNNaker channels. The packet FIFOs will stop accepting packets when FULL, which triggers a nack frame with the current frame sequence number. The flow control mechanisms described in Section III-D attempt to avoid frame rejections and the need for retransmissions.

C. PACKET STORE OPERATION

The packet store is a key component as it allows the efficient use of HSSL bandwidth when retransmitting a frame: the packets present in the original frame are included again but, in addition, packets that arrived after the sending of the original frame can be included in the retransmitted one. Fig. 10 shows a PS block diagram. The FIFO storage is logically split in three areas. The first area, read/not acknowledged (RNA), holds packets that have already been sent across the HSSL and are waiting to be acknowledged, the second, written/not read (WNR), holds packets that are waiting to be sent and, finally, there is an EMPTY area where new packets can be stored. PS keeps a pointer associated with each area (ack\(_p\), read\(_p\) and write\(_p\), respectively), that are updated as the result of the operations described below. Correct PS operation is supported by the sequence → reg MAP, also shown in Fig. 10. The MAP keeps track of the read\(_p\) position associated with every frame in-flight and is used to update both read\(_p\) and ack\(_p\).

The MAP operation is described in Scheme 1. When a packet in WNR is used in a new assembled frame, read\(_p\) is updated and the packet logically moves to RNA. Every PS records the value of its read\(_p\), even if no packet from that PS is present. This is required to allow the inclusion of newly arrived packets in a reassembled frame. The frame sequence number is used as the MAP write address. When an ack frame arrives, ack\(_p\) is updated past the ack sequence number, flushing RNA packets, i.e., logically moving RNA slots to EMPTY. If a nack frame arrives, read\(_p\) is pulled back to the value corresponding to the nack sequence for retransmission. Also, a nack carries an implicit ack for every previous frame, thus ack\(_p\) is updated to the sequence number MAP value.

Scheme 1 Packet Store MAP Operation

1: if frame assembled then
2: MAP[frame_sequence] = read\(_p\)
3: if ack received then
4: ack\(_p\) = MAP[ack_sequence] + 1
5: if nack received then
6: read\(_p\) = MAP[nack_sequence]
7: ack\(_p\) = MAP[nack_sequence]

In this scheme, a new frame can be assembled concurrently with the arrival of either an ack or a nack. It is clear from the scheme that assembling a frame simultaneously with the arrival of a nack causes concurrent read and write accesses to both read\(_p\) and the MAP. As naks happen infrequently, if at all, spinNNlink uses Scheme 2, a less concurrent approach, that makes naks exclusive and simplifies the logic, with negligible impact on bandwidth.

Scheme 2 Simplified PS MAP Operation

1: if nack received then
2: read\(_p\) = MAP[nack_sequence]
3: ack\(_p\) = MAP[nack_sequence]
4: else
5: if frame assembled then
6: MAP[frame_sequence] = read\(_p\)
7: if ack received then
8: ack\(_p\) = MAP[ack_sequence] + 1

D. HIGH-SPEED SERIAL LINK FLOW CONTROL

As in most interconnect systems, spinNNlink requires some form of flow control. Finite storage capacity in PD means
that, in the presence of congestion in any outgoing SpiNNaker channels, the packet FIFOs may get full and stop accepting packets. This results in rejected frames and the need for retransmissions, with a negative impact on bandwidth utilization and energy consumption. spiNNlink uses credit-based flow control [13] to limit the number of unacknowledged, transmitted frames to guarantee that PD can always accept them. FA consumes credit with every assembled frame and is re-credited by the remote PD based on the occupancy of its FIFOs.

Unfortunately, due to the independent operation of the SpiNNaker channels and their unpredictable (spike) traffic patterns, a global credit value for the HSSL is inefficient, as the PD would have to re-credit FA based on the packet FIFO with fewer available slots. A congested SpiNNaker channel would affect all others and, in the worst case, could deadlock the HSSL. A common solution to this problem is to keep a separate credit value for every channel, with the ensuing impact on design area and bandwidth utilization.

An alternative to expensive per-channel credit is to use per-channel flow control (cfc), i.e., congested channels are disabled to prevent them from affecting others, re-enabling them once the congestion has eased. A basic cfc scheme uses Xon/Xoff control on each channel, based on the occupancy of its PD FIFO, and Low and High Water Marks (LWM/HWM). In this scheme, channel \( i \) is disabled (PD sends \( Xoff \)) when its FIFO occupancy \( \text{occ}_i \) rises above HWM and is re-enabled (PD sends \( Xon \)) when it drops below LWM. The two marks establish the required hysteresis to prevent oscillation. Although this scheme avoids the impact of congested channels, frame retransmissions can still occur due to the time it takes the \( Xon/Xoff \) controls to traverse the HSSL.

spiNNlink uses a combination of global credit and per-channel flow control to guarantee that PD always has available FIFO space for any frame in flight, avoiding retransmissions and the negative effects of congested channels. PD re-credits the remote FA based on the occupancy of the enabled FIFOs, ignoring the disabled ones. The operation of each channel is shown in Scheme 3. When the occupancy of an enabled channel rises above HWM the channel is disabled. As a result, global credit may increase so it is recomputed. The key insight is that a disabled channel is re-enabled only when it will not cause a drop in global credit, as this could lead to frame rejection.

As explained in Section III-C, PD sends frame \( acks \) to the remote FA to confirm frame arrival, allowing it to evict packets kept in PS for possible retransmission and, thus, freeing space to accept new packets. To improve bandwidth utilization in this new scheme, PD also uses frame \( acks \) to re-credit the remote FA. In spiNNlink, FA uses the difference in sequence numbers of consecutive \( acks \) to increment its current global credit. This mechanism is safe even if \( acks \) are dropped or duplicated as a result of errors or retransmissions. For this overloading of the \( ack \) function to work correctly, \( acks \) must be sent when packets exit PD FIFOs, creating free space, as opposed to when they are stored. This may delay packet entry into the remote FA but not its sending across the HSSL, as FA is still re-credited at the earliest opportunity. As with cfc, PD sends \( acks \) to FA on \( data \) frames (field \( ack \) in Fig. 7) with negligible impact on HSSL bandwidth.

### IV. spiNNlink - SpiNNaker INTERFACE

Interfacing with the asynchronous, handshake-based SpiNNaker channels poses a throughput challenge for spiNNlink. Fully clock-less implementations of the interface [11], [14], which avoid the need for synchronization have shown promise but are not usually a good match for FPGA implementation and were not considered for spiNNlink. Fig. 11 shows a traditional interface implementation, with data flowing from a spiNNlink link sender, on the left, to the SpiNNaker chip, on the right. The channel uses the 2-of-7 code and handshake mechanism described in Section III.

![FIGURE 11. spiNNlink to SpiNNaker channel interface (traditional scheme).](image-url)

The communications throughput is limited by the latency around the loop: a new 4-bit packet flit \((\text{flit}_\text{data} + \text{flit}_\text{vid})\) arrives at the sender input, is encoded and sent to SpiNNaker where the input pipeline register (pr) captures and acknowledges it. The \( ack \) signal comes back and is synchronized to the spiNNlink sender clock through two FFs. Finally, the sender indicates its readiness to accept a new flit \((\text{flit}_\text{rdy})\). With this implementation, the spiNNlink side of the interface imposes a lower bound of 3 sender clock cycles on the loop time, but it may be higher if the SpiNNaker chip takes longer than 1 sender clock cycle to respond.

Predictive handshaking [15], an alternative interface strategy, is based on the notion that, even though the channels are asynchronous, they accept flits at a nearly-constant pace.
FIGURE 12. spiNNlink to SpiNNaker channel interface (STAC scheme).

The interface has a configuration step in which it determines the pace of the handshake and uses it to predict the arrival of each handshake acknowledge, treating it as a synchronous signal. In this scheme, the sender sends flits to the channel at periodic intervals and the ack signal is processed in parallel, keeping it outside the critical path, thus providing increased bandwidth. Unfortunately, Predictive handshaking is not designed to deal with asynchronous back pressure, i.e., situations in which the asynchronous channel stalls for an unbounded time due to traffic congestion.

Fig. 12 shows a novel strategy developed for spiNNlink that uses Synchronous Timing and Asynchronous Control (STAC) to guarantee correct operation. Scheme 4 summarizes STAC operation. The sender goes through sync and async mode phases. As in Predictive Handshaking, the link sender treats the handshake acknowledge as a synchronous signal when operating in sync mode. At the point where SpiNNaker can apply back pressure, the sender switches to async mode, where it treats the handshake acknowledge as an asynchronous signal, and completes a fully-asynchronous handshake. The sender identifies when it reaches the back-pressure point (BPP) using the flit and ack counters.

Scheme 4 STAC Operation

1: if MODE == sync then
2: send data
3: if count == BPP then
4: MODE = async
5: else
6: wait-for ACK
7: send data
8: MODE = sync

Establishing the BPP requires understanding of the SpiNNaker operation which, as shown in Fig. 11 has a packet buffer and will only apply back pressure when the buffer and all the pipeline registers (pr) become full.

V. spiNNlink ASSESSMENT

spiNNlink is in operation in the million-core SpiNNaker machine and over 5 million simulation jobs have already been completed. Job sizes range from 1 SpiNNaker board to the complete machine, comprising 1,200 boards. The novel aspects of the implementation have proven correct and reliable, as discussed in the following sections.

A. FPGA RESOURCE UTILIZATION

spiNNlink was developed for the Xilinx Spartan-6 FPGA using the Linux version of ISE v14.7, the Xilinx-recommended tool for Spartan designs. Table 1 lists the utilization of the different FPGA resources as computed by ISE. The cost-effective Spartan-6 device was chosen for its HSSL support but also for its input/output pin count, a good match for the large number of connections to the SpiNNaker chips, reflected in the high utilization of Input/Output buffers (IOB).

<table>
<thead>
<tr>
<th>Device</th>
<th>Used</th>
<th>Available</th>
<th>Utilization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice Registers</td>
<td>15026</td>
<td>54576</td>
<td>27%</td>
</tr>
<tr>
<td>Slice LUTs</td>
<td>19522</td>
<td>27288</td>
<td>71%</td>
</tr>
<tr>
<td>Fully-used LUT-FF pairs</td>
<td>11431</td>
<td>23117</td>
<td>49%</td>
</tr>
<tr>
<td>Bonded IOBs</td>
<td>281</td>
<td>296</td>
<td>94%</td>
</tr>
<tr>
<td>BUFG/BUFGCTRLs</td>
<td>5</td>
<td>16</td>
<td>31%</td>
</tr>
</tbody>
</table>

B. PERFORMANCE

An efficient communications infrastructure is key to simulating spiking neural networks in real time as, in most cases, communication costs dominate performance and scale non-linearly with neural network size [1]. Table 2 shows spiNNlink relevant performance data. For reference, available data for InfiniBand [16] is also shown, although the figures are not directly comparable as InfiniBand is a switched fabric while spiNNlink is a point-to-point interconnect.

<table>
<thead>
<tr>
<th>units</th>
<th>spiNNlink</th>
<th>InfiniBand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Effective data throughput</td>
<td>Gbit/s</td>
<td>2.16</td>
</tr>
<tr>
<td>Node latency</td>
<td>ns</td>
<td>665</td>
</tr>
<tr>
<td>Power consumption (per FPGA)</td>
<td>W</td>
<td>1.67</td>
</tr>
<tr>
<td>SpiNNaker Bisection bandwidth</td>
<td>Gbit/s</td>
<td>480</td>
</tr>
<tr>
<td>SpiNNaker worst-case latency</td>
<td>μs</td>
<td>34 – 46</td>
</tr>
</tbody>
</table>

The table shows that spiNNlink provides a 2.16 Gbit/s effective HSSL data throughput. This is achieved by running the HSSL at 3.0 Gbit/s using an 8b/10b encoding (80% efficiency), and under worst-case traffic, i.e., maximum data frame HSSL occupancy (90% efficiency). Although Fig. 7 may suggest that 32-bit frames with 8-bit headers are inefficient, this only applies to infrequently-transmitted control frames. Routine control is carried by data frames as a constant overhead. Data frames have the important property that data-to-overhead ratio increases with higher network traffic, providing higher efficiency when needed the most.

Table 2 also shows the bisection bandwidth for the million-core system and its worst-case latency, across
128 network nodes and 8 spiNNlinks. Both parameters comfortably meet the requirements for most real-time neural simulations. The cortical microcircuit model, introduced earlier, represents a significant challenge due to its complexity in terms of number of neural devices but also in terms of its high connectivity and demanding traffic. Table 3 presents the relative performance and energy consumption, reported in [1], of cortical microcircuit simulations run on three different platforms: SpiNNaker, a GPU-based system and a high-performance computer.

**TABLE 3.** Brain cortical microcircuit simulation (relative performance).

<table>
<thead>
<tr>
<th></th>
<th>SpiNNaker</th>
<th>GPU</th>
<th>HPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time simulation slowdown</td>
<td>1</td>
<td>2x</td>
<td>3x</td>
</tr>
<tr>
<td>Energy per synaptic event (μJ)</td>
<td>0.628</td>
<td>1x</td>
<td>10x</td>
</tr>
</tbody>
</table>

The table shows that Spinnaker achieves true real-time simulation while the GPU and HPC systems require twice and three times longer than real time, respectively. It is interesting to note that the SpiNNaker simulation already operates across multiple boards, thus not facing the communication bottleneck that may hinder HPC and GPU performance as simulations scale up. Additionally, SpiNNaker and the GPU system show similar energy consumption while the consumption of the HPC system is an order of magnitude higher.

**C. RELIABILITY**

We tested spiNNlink using synthetic applications designed to stress the interconnect of the complete SpiNNaker system. In these tests, a sender subset of the cores on every chip send packets to a receiver subset of cores on all its neighboring chips. The cores report transmitted and received packets as well as the achieved throughput. Statistics on transmitted, received and rejected (nack) HSSL frames, as well as frame errors, are collected in the spiNNlink register bank.

Our tests, complemented with statistics collected over months of SpiNNaker operation, show that HSSL error rates are negligible. After correct SATA cable resitting and using adequate GTP parameter settings, most channels have never reported an error, as they usually operate for less than a week per job, not long enough for errors to manifest. Worst-case links show under $10^{-5}$ errors per second (BER < $3.3 \times 10^{-15}$), with imperceptible impact on bandwidth. When frame errors did manifest, the frame retransmission mechanism insured that no packets were lost on spiNNlinks, resulting in 100% correct transmission on all tests. In some test scenarios, SpiNNaker routers dropped packets due to congestion, but this is correct SpiNNaker behavior.

**D. ASYNCHRONOUS INTERFACE THROUGHPUT**

Fig. 13 shows a histogram of the throughputs achieved by the 1,152 spiNNlink-to-SpiNNaker channels in the same 24-board card frame used for Fig. 3.

Two different measurements are presented: when spiNNlink uses the traditional synchronization scheme the channels show throughputs in the 109–120 Mbit/s range, a significant mismatch to the chip-to-chip channels shown in Fig. 3. spiNNlink using the novel STAC interface demonstrates a doubling of the achieved throughputs, occupying the 205–220 Mbit/s range.

**E. COMPLEMENTARY USES**

spiNNlink cooperated in the complex and error-prone task of cabling the million-core SpiNNaker system, which required 3,600 SATA cables for board-to-board interconnect. spiNNlink idle frames carry a programmable 16-bit value that is stored in diagnostic registers. These values were used to verify the correct installation of each cable in real time, ensuring that mistakes were highlighted and fixed immediately. A video showing this process for a half-size SpiNNaker machine is available online [17].

spiNNlink also helps during operation as most SpiNNaker jobs only require a subset of the boards in the machine. When a job is requested, the job allocation software turns off the spiNNlinks on the periphery of the allocated boards to provide job isolation, i.e., insures that concurrent jobs can not interfere with each other.

**F. PUSHING spiNNlink FORWARD**

spiNNlink is part of spI/O [18], the open-source library of FPGA modules for SpiNNaker Input/Output. The design is being pushed forward in the context of a research project focusing on exascale computing, which has extremely high bandwidth demands. Table 4 lists some of the key improvements achieved so far, impressively with no significant changes to the logic. The new implementation basically requires changes to allow data streams to be interleaved onto the HSSL bonded channels, not used in the original design.
TABLE 4. spiNNlink vs exascale project.

<table>
<thead>
<tr>
<th>FPGA device</th>
<th>spiNNlink</th>
<th>exascale project</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA HSSL IP</td>
<td>Spartan-6</td>
<td>Zynq UltraScale-</td>
</tr>
<tr>
<td></td>
<td>GTP</td>
<td>GTH</td>
</tr>
<tr>
<td>Effective data throughput [Gbit/s]</td>
<td>2.16</td>
<td>51.2 – 64.0</td>
</tr>
<tr>
<td>Channel bonding</td>
<td>1x</td>
<td>4x</td>
</tr>
<tr>
<td>Encoding</td>
<td>8b/10b</td>
<td>8b/10b – 64b/66b</td>
</tr>
</tbody>
</table>

VI. SUMMARY AND CONCLUSIONS

The million-core SpiNNaker computer is capable of running brain cortical simulations in true real-time, not currently possible using conventional HPCs or GPUs. Providing SpiNNaker with an efficient interconnect was a big challenge due to its massive scale but also because of its unconventional, clock-less fabric optimized for small multicast packets. Rather than using ill-matched conventional technologies, we decided to develop a bespoke, FPGA-based high-speed interconnect. The chosen FPGA proved a good development platform as well as an efficient production device, given that its support for the physical layer of the high-speed link is of high quality and is well documented.

The first challenge faced was that, in general, interfacing FPGAs to asynchronous, handshake-based channels incurs a long latency, with the corresponding throughput penalty. To avoid this penalty, we developed an alternative interface that, based on the SpiNNaker channel operation, predicts correctly the arrival of handshake signals and provides twice the throughput of the traditional scheme.

Further challenges were related to the efficient use of the available high-speed link bandwidth when multiplexing eight SpiNNaker channels across the link. We developed a novel, arbitration-free, reliable frame transport protocol with a flow control mechanism that avoids the negative impact of congested channels when using global credit and the excessive penalty of per-channel credit. Its implementation required a variable-size, bandwidth-efficient frame format, a modified FIFO operation for correct credit handling, and an overloaded function of frame acknowledges, and a novel packet store.

We conclude, from experimental results and months of SpiNNaker operation, that developing a bespoke, FPGA-based high-speed interconnect was challenging but also rewarding, cost effective and an efficient way to meet SpiNNaker throughput and latency requirements.

ACKNOWLEDGMENT

SpiNNaker has been 20 years in conception and 15 in construction, and many folk in Manchester and in our many university and industry partners across the EU and beyond have contributed to its success. Aspects of the spiNNlink design, in particular, benefited from a collaboration with colleagues from Instituto de Microelectrónica de Sevilla (IMSE-CNMI, Spain). We gratefully acknowledge all of these contributions. Finally, we are grateful to the anonymous reviewers of the manuscript for their insightful comments and recommendations.

REFERENCES

JIM GARSIDE received the B.Sc. degree in physics and the Ph.D. degree in computer science from The University of Manchester, Manchester, U.K., in 1983 and 1987, respectively. His doctoral work looked at digital signal processing architectures. He spent some time working on parallel architectures with Transputers. After a brief sojourn in the software industry, he returned to The University of Manchester as a Lecturer, in 1991. Later research work has primarily been concerned with VLSI technology, particularly with the Amulet asynchronous ARM processors, and more recently, with SpiNNaker. His current interest includes power-efficient processing especially using hardware reconfiguration.

LUIS A. PLANA received the B.Sc. degree from Universidad Simón Bolívar, Caracas, Venezuela, and the Ph.D. degree in computer science from Columbia University, New York, NY, USA. He worked with Universidad Politécnica, Venezuela, for over 20 years, where he was a Professor of electronic engineering. He is currently a Research Fellow with the Department of Computer Science, University of Manchester, Manchester, U.K. His research interests include neuromorphic systems engineering and energy-efficient many-core systems architecture.

JEFFREY PEPPER received the Diploma degree in electronic engineering and computer technology and the B.Sc. degree (Hons.) in engineering from Open University, Milton Keynes, U.K. He joined as the Technical Staff with the Department of Computer Science, The University of Manchester, Manchester, U.K., in the mid-1980s. He is currently a Senior Experimental Officer specializing in CAD tools, chip design methodologies, and hardware description languages.

SIMON DAVIDSON received the Ph.D. degree from the University of Sheffield, Sheffield, U.K., in 2000, with a focus on neural associative memory. He spent a decade as a Microprocessor and the ASIC Designer with companies including SGSThomson, Bristol, U.K., and Hewlett Packard, Grenoble, France and finally with ARC International, Elstree, U.K., where he led the development of the ARC Tangent-A5 processor and co-designed the ARCompact instruction set. He is currently a Research Fellow with The University of Manchester, Manchester, U.K. His current research interests include spiking neural memory and the implementation of massively parallel symbol processing systems on SpiNNaker.

STEVE FURBER (Fellow, IEEE) received the B.A. degree in computer science and the Ph.D. degree in research into local area networks from the University of Cambridge, Cambridge, U.K., in 1980 and 1984, respectively. He was subsequently a Research Fellow with the Computer Laboratory, University of Cambridge, a self-employed Computer Consultant, from 1986 to 1993, and a Research Fellow with the Department of Computer Science, University of Manchester, Manchester, U.K., from 1993 to 2017. He is currently a Senior Engineer with Mindtree, Manchester.

MIKEL LUIJÁN received the Ph.D. degree in computer science from The University of Manchester. He was a Royal Society University Research Fellow with the Department of Computer Science, University of Manchester. He is currently a Professor with the Department of Computer Science, The University of Manchester, where he holds the ARM/Royal Academy of Engineering Research Chair of computer systems. His research interests include managed runtime environments and virtualization, manycore architectures, and application-specific systems optimizations.

JEFFREY PEPPER received the Diploma degree in electronic engineering and computer technology and the B.Sc. degree (Hons.) in engineering from Open University, Milton Keynes, U.K. He joined as the Technical Staff with the Department of Computer Science, The University of Manchester, Manchester, U.K., in the mid-1980s. He is currently a Senior Experimental Officer specializing in CAD tools, chip design methodologies, and hardware description languages.

STEVE FURBER (Fellow, IEEE) received the B.A. degree in computer science and the Ph.D. degree in research into local area networks from the University of Cambridge, Cambridge, U.K., in 1980 and 1984, respectively. He was subsequently a Research Fellow with the Computer Laboratory, University of Cambridge, a self-employed Computer Consultant, from 1986 to 1993, and a Research Fellow with the Department of Computer Science, University of Manchester, Manchester, U.K., from 1993 to 2017. He is currently a Senior Engineer with Mindtree, Manchester.