SpiNNaker—Programming Model

Andrew D. Brown, *Senior Member, IEEE*, Steve B. Furber, *Fellow, IEEE*, Jeffrey S. Reeve, *Senior Member, IEEE*, Jim D. Garside, Kier J. Dugan, Luis A. Plana, *Senior Member, IEEE*, and Steve Temple

Abstract—SpiNNaker is a multi-core computing engine, with a bespoke and specialised communication infrastructure that supports almost perfect scalability up to a hard limit of $2^{16} \times 18 = 1{,}179{,}648$ cores. This remarkable property is achieved at the cost of ignoring memory coherency, global synchronisation and even deterministic message passing, yet it is still possible to perform meaningful computations. Whilst we have yet to assemble the full machine, the scalability properties make it possible to demonstrate the capabilities of the machine whilst it is being assembled; the more cores we connect, the larger the problems become that we are able to attack. Even with isolated printed circuit boards of 864 cores, interesting capabilities are emerging. This paper is the third of a series charting the development trajectory of the system. In the first two, we outlined the hardware build. Here, we lay out the (rather unusual) low-level foundation software developed so far to support the operation of the machine.

Index Terms—Interconnection architectures, parallel processors, neurocomputers, real-time distributed

1 Introduction

SPINNAKER is a multi-core message-passing computing engine based upon a completely different design philosophy from conventional machine ensembles. It possesses an architecture that is completely scalable to a limit of over a million cores, and the fundamental design principles disregard three of the central axioms of conventional machine design: the core-core message passing is non-deterministic (and may, under certain conditions, even be non-transitive); there is no attempt to maintain state (memory) coherency across the system; and there is no attempt to synchronise timing over the system.

Notwithstanding this departure from conventional wisdom, the capabilities of the machine make it highly suitable for a wide range of applications, although it is not in any sense a general purpose system: there exists a large body of computational problems for which it is spectacularly *ill*-suited. Those problems for which it *is* well-suited are those that can be cast into the form of a *graph* of *communicating entities*. The flagship application for SpiNNaker—neural simulation—has guided most of the hard architectural design decisions, but other types of application—for example mesh-based finite difference problems—are equally suited to the specialised architecture.

The hardware architecture of the machine is described in detail elsewhere [1], [2], [3], [4], [16]—here we describe the low-level software infrastructure necessary to underpin the operation of the machine. It is tempting to call this an operating system, but we have resisted this label because the term induces preconceptions, and the architecture and mode of operation of the machine does not provide or utilise resources conventionally supported by an operating system. Each of the million (ARM9) cores has—by necessity—only a small quotient of physical resource (less than 100 kbytes of local memory and no floating-point hardware). The inter-core messages are small (<= 72 bits) and the message passing itself is entirely hardware brokered, although the distributed routing system is *controlled* by specialised memory tables that are configured with software. The boundary between soft-, firm- and hardware is even more blurred than usual.

SpiNNaker is designed to be an event-driven system. A packet arrives at a core (delivered by the routing infrastructure), and causes an interrupt, which causes the (fixed size) packet to be queued. Every core polls its incoming packet queue, passing the packet to the correct packet handling code. These packet event handlers are (required to be) small and fast. The design intention is that these queues spend most of their time empty, or at their busiest, containing only a few entries. The cores react quickly (and simply) to each incident packet; queue sizes much larger than one are regarded as anomalous (albeit sometimes necessary). If handler ensembles are assembled that violate this assumption, the system performance rapidly (and uncompetitively) degrades.

The components of this paper are as follows:

- In Section 2, we review—selectively—existing multi-core and neuromorphic activities.
- Section 3 highlights the differences between SpiNNaker and a conventional architecture.
- Section 4 contains a précis of the SpiNNaker hardware architecture. Most of the material in this section has appeared in [1], but some aspects are enhanced.
- In Section 5 we describe the bootstrapping, initialisation and low-level kernel software.
- Section 6 contains an outline of the programming environment provided to support the interrupt

- *S.B. Furber, J.D. Garside, L.A. Plana, and S. Temple are with the School of Computer Science, The University of Manchester, Manchester, England M13 9PL, United Kingdom.*
  E-mail: {sbf, jgarside, plana, temples}@cs.man.ac.uk.
- *A.D. Brown, J.S. Reeve, and K.J. Dugan are with the Department of Electronics and Electrical Engineering, The University of Southampton, Southampton, Hampshire, United Kingdom.*
  E-mail: {adb, jsr, kjd1v07}@ecs.soton.ac.uk.

Manuscript received 1 Aug. 2013; revised 17 Mar. 2014; accepted 18 May 2014. Date of publication 0 . 0000; date of current version 0 . 0000.
Recommended for acceptance by M. Guo.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2014.2329686

handlers, and an exemplar of how one might perform meaningful computation within the framework provided by SpiNNaker. The data structures supporting the mapping of a simple system onto a tiny processor mesh are described. We also provide an overview of how system output is realised.
- Finally, we mention some of the many future challenges we have to address in exploiting this machine.

We have *not* described the offline support tool portfolio or any quantitative measurements—these, and the other physical domains for which SpiNNaker *is* ideally suited will all be described at a later date.

1.1 Terminology

Within the context of this paper, some terms would benefit from a prior introduction/definition:

- An individual ARM9 processor (plus associated local resources) is a **core**.
- The cores are physically implemented (in UMC 130 nm silicon), 18 to a die. The die also contains the routing engine, and physically mounted on top of it (stitch-bonded) within the same package is 128 Mbyte of SDRAM. This entire structure is a **node**, 2^{16} of which are connected together to form the **SpiNNaker engine**. The node boundaries (necessary but an artefact of fabrication) are transparent to the connected mesh of cores. Phrases such as "*processor topology*" and "*core graph*" refer to the **physical (functioning) hardware mesh of cores**. 2^{16} is a hard limit—the internal node address uses only 16 bits.
- SpiNNaker is a computing engine that comes into its own with programming problems that can be coerced into the form of a mesh, or graph, of communicating entities. In order to work, this abstract **problem graph** must be mapped onto the physical core graph. This mapping is many:1, and is the responsibility of the initialisation software.
- The vertices of the core graph are—naturally enough—cores, and the vertices of the problem graph are referred to generically as **(problem) devices**. As will be seen later, the set of behaviours embodied by a device are broad and eclectic, realised as small fragments of code running on the core to which the device has been mapped.

2 The Multicore/Neuromorphic Landscape

Building large hardware is extremely costly, from the point of view of both money and manpower, and most 'broad-scale' multi-core research is undertaken by industrial sponsors. (SpiNNaker is unusual in that the entire design effort was undertaken in University research groups.) However, the "unconventional-architecture" landscape is not entirely unpopulated:

- *Anton* [5] is a special-purpose supercomputer consisting of 512 custom ASICs arranged in a high-bandwidth 3D torus network designed for simulating molecular dynamics (MD) problems.
- Intel have produced a prototype chip that features 48 Pentium-class IA-32 processors, arranged in a 2D 6×4 grid network optimised for the message passing interface [6].
- *Centip3De* is a 130 nm stacked 3D near-threshold computing (NTC) chip design that distributes 64 ARM Cortex-M3 processors over four cache/core layers connected by face-to-face interface ports [7].
- Satpathy et al. [8] present a 128 bit 64-input 64-output single-stage swizzle-switch network (SSN) which is similar to a crossbar switch but also supports multicast (MC) messages.
- *TILE64* is a chip-multiprocessor architecture design that arranges 64× 32 bit VLIW processors in a 2D 8×8 mesh network that supports multiple static and dynamic routing functions [9].
- BlueBrain [10] is not an unconventional architecture, but the software organisation does contain parallels to SpiNNaker. The simulator used by BlueBrain, NEURON, is distributed over up to 128K processors (each with 512 MB of RAM), with coarse communications supported by MPI.

As the size of parallel systems increases, the *proportion* of resource consumption (including design effort) absorbed by 'non-computing' tasks (communications and housekeeping) increases *disproportionally*. Architectures that sidestep these difficulties with unconventional mechanisms are gaining traction in specialised areas. SpiNNaker is designed to be effective for the simulation of systems comprising many simple elements with a massive communications component.

Other examples of massively-parallel neurally-inspired architectures include:

- NeuroGrid [11] is an example of an analogue implementation of a neural equation solver with digital communications that operates in biological real time by virtue of using sub-threshold analogue circuits.
- The high input count neural network (HICANN) chip [12], developed within the EU FACETS project, uses above-threshold analogue circuits to deliver large-scale neural models that run 10,000 times faster than their biological equivalents; a technology that has been carried forward through the EU BrainScaleS project to form a major neuromorphic computation platform in the EU Human Brain Project (alongside SpiNNaker).
- IBM has demonstrated a digital neural accelerator chip [13] with the specific objective of achieving deterministic and consistent behaviour between the software model and the silicon.

Many of these concepts can be traced back to the original analogue neuromorphic work at Caltech by Mead [14].

3 Principles of Use

3.1 Anatomy of a Conventional Parallel Program

The anatomy of a *conventional* parallel program is well known. The program designer can realistically expect a host of system level resources to be made available, and designs a set of arbitrarily complicated *programs*, the intercommunication choreography of which may itself be extremely complex.

The messages by which these processes communicate are made up of an arbitrary number of *units*, the structure of

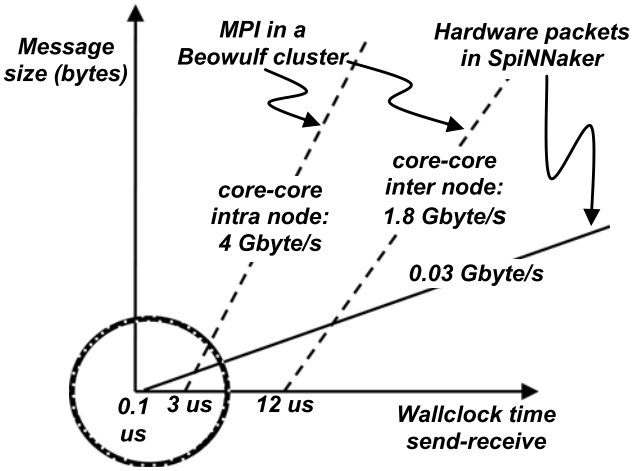


Fig. 1. Temporal cost of message passing.

which may be defined by the program designer. The temporal cost of sending a message is usually a function of the message size: Fig. 1.

3.2 Anatomy of a SpiNNaker Program

In contrast, the anatomy of a SpiNNaker-based parallel program is shown in Fig. 2. The structure (topology) of the problem graph is distributed throughout the route tables of the hardware—thereafter the potential routes of all the messages are considered fixed throughout the program execution. (It is possible to change the routing information during program execution, but this is an expensive and hard task.)

An incoming message to a node causes an interrupt to be generated in a core, which is handled by an appropriate (user supplied) fragment of code (the *interrupt handler*). This handler in turn may or may not cause consequent messages to be sent. Two points are of note here:

- A handler says "send message", but has no control where the outgoing message goes (that information is distributed throughout the routing table). An incoming message contains the information outlined in Table 1, and the incident (delivery) port is visible to the handler, but the route across the interconnect fabric is not available to the handler.
- A message is launched, propagated and delivered with a delay dictated by the ambient hardware traffic on the route. It contains no timestamp of any sort; the interrupt handler is entirely asynchronous and reactive.

SpiNNaker as a simulation engine operates at a much finer (and non-hierarchical) level of granularity than conventional simulators. In a conventional (electronic) system description—say, VHDL or Verilog-based—the floorplan interconnect is (relatively speaking) uninteresting—the complexity lies *inside* the component descriptions. In SpiNNaker, the component descriptions are (relatively speaking) very simple—the complexity resides in the interconnect topology *between* the problem devices.

The behaviour of the problem devices is realised by the interrupt handler code, which is supplied by the user, and can, of course, be arbitrarily complex, but supplying large and complex handlers moves the system out of its intended functional design space, and the performance will suffer enormously.

4 Hardware Overview

4.1 Architecture

SpiNNaker is a homogeneous network of triangularly connected **nodes**, as in Fig. 3. The mesh—shown planar in the figure—has its opposing edges identified with each other, so the whole 'computing surface' is effectively mapped to the surface of a toroid. (Many other mappings produce an equivalent effect.) Each node corresponds to a physical chip, and contains an Ethernet controller implemented in silicon. In principle, an arbitrary number of these may be connected to external (conventional) machines via an external Ethernet. The internal structure of each node is outlined in Fig. 4. The essential components are the set of eighteen ARM9 **cores**, the **message router**, watchdog timers/counters, all interconnected via the **node NoC**. All the cores in the entire system have a 32-bit memory space; portions of the individual maps refer to different tranches of physical memory. The full details of the memory map may be found in [1], but it is useful to review some relevant aspects here:

- Each *node* contains **128 M SDRAM** and **32 k SRAM**—this is referred to as *node-local* memory.
- Each *core* contains **64 k DTCM** (data memory) and **32 k ITCM** (instruction memory)—this is referred to as *core-local* memory, and provides a Harvard execution model for each individual core.
- Each *node* also contains a 32k (memory mapped) BOOT ROM.

The essentially homogenous nature of the coarse interconnect (Fig. 3) allows the size of the overall machine to be almost arbitrary; the only constraint with the current design being the size of the address space used to identify the nodes (currently this is 16 bits, giving a maximum node count of 2^{16}). The nodes are assembled onto PCBs holding 48 nodes each; each PCB dissipates around 20 to 50 W depending on workload. When fully assembled, the system will contain a maximum of 65,536 (2^{16}) nodes, giving a total possible core count of $65{,}536 \times 18 = 1{,}179{,}648$, with over 8.5 Tbyte of on-board distributed memory. It will dissipate around 90 kW under full computational load.

The boards form another artificial boundary. The board-to-board interconnect is supported by three Xilinx Spartan-6 FPGAs mounted on each board; again, these have a broad mandate to be transparent to the core-core communications.

4.2 The Message-Passing Infrastructure

Although the cores on a given node may communicate with each other via shared memory [1], the dominant communication route between cores—and the *only* route between cores on different nodes—is by message passing.

Message passing on conventional cluster machines is expensive. Fig. 1 shows the approximate message latency and throughput times measured on a 1,000+ core Beowulf cluster machine[1], using MPI brokered by Myrinet [15]. It has

1. 1008 compute nodes, 2 × 4 core 2.27GHz Nehalem processors (i.e. 8 processors/node) providing > 72 TFLOPS.

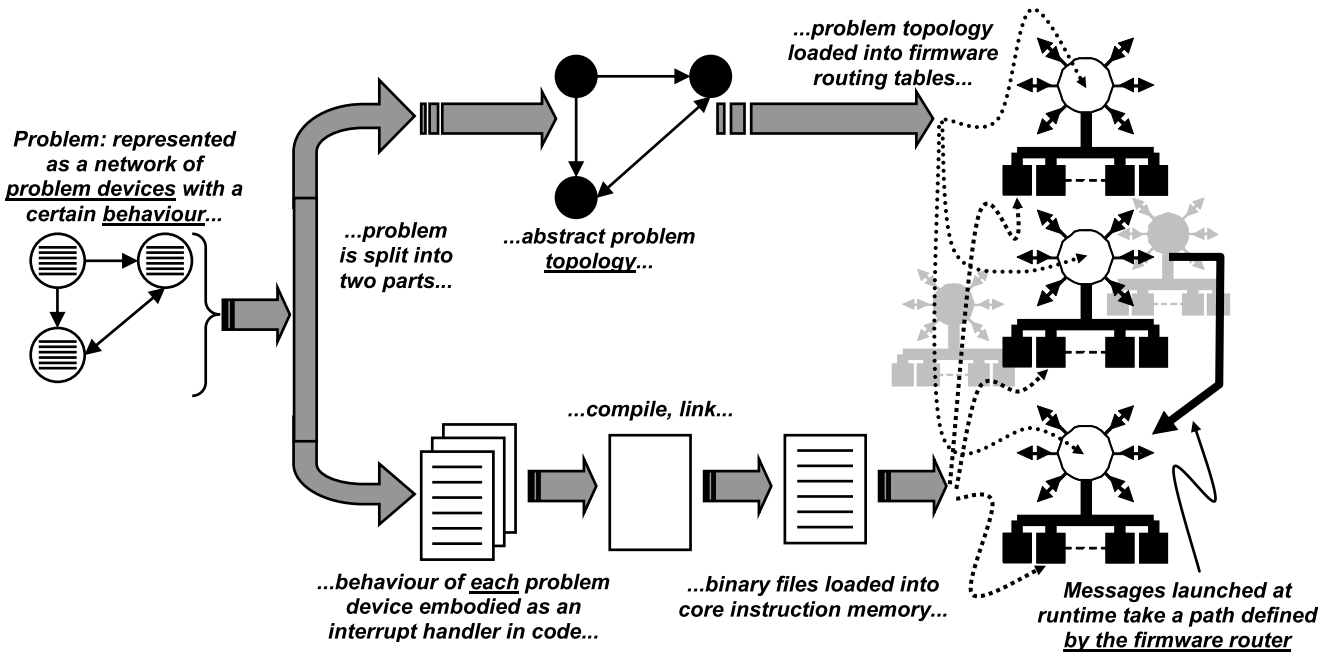


Fig. 2. The anatomy of a SpiNNaker parallel program.

been reported [16] that some biological simulation codes spend over 30 percent of their wall-clock time sending and receiving messages.

Messages in SpiNNaker are hardware brokered, moving through the communications fabric controlled by the router subsystems in each node. The size of a message is fixed at 72 bits (it is hardware), and each step (node-node transit) takes around 0.1 μs. Thus in the complete machine, configured as a toroid, the *maximum* node-node hop delay (when the chosen nodes are on opposite sides of the torus) is $\sqrt{(2^{16})}/2 * 0.1 \sim 12.8$ μs. The *minimum* transit time (two cores on the same node) is 0.1 μs. In every case the individual message throughput is around 30 Mbytes/s. By the standards of today, this is not a high number, but factored into the interconnect topology gives the machine as a whole a bisection bandwidth of around 4.8 Gpackets/s.

SpiNNaker comes into its own when a problem can be cast into a form that requires many, many tiny asynchronous messages—the region near the origin in Fig. 1—and there are a diverse and interesting set of problems that meet this criterion.

From the perspective of the nodes, SpiNNaker is indeed a homogeneous, isotropic computing mesh. However, *within* a node, all the cores are not equal. On power-up, a (designed) race elects one core as the **monitor** core. This core—identified as core 0 by definition—then interrogates its node-local peers, assigning them identifiers 1 . . . 16. (Represented internally by 4 bits—we can do this because the monitor core is special on a number of levels, and is never—can never be—addressed by the same mechanism as an application core.) These become the **application** processors.

The low-level fault tolerance philosophy is detailed in [1]. One of the early design decisions taken made the assumption that it would be naive—in a system consisting of over 65,000 chips—to assume that we could rely on 100 percent yield. On power-up, the cores self-organise into one monitor core and (up to) 16 functioning application cores, with a core to spare. We have so far taken delivery of around 750 chips, of which 82 percent had at least 17 functioning cores. The self-organising initialisation is capable of configuring nodes with any number of failed cores, although of course the definition of functioning is not all-embracing. (We have one core that resolutely refuses to do anything whatsoever *except* report that it is functioning correctly.) Any node with at least two functioning cores is considered useful.

Message transmission is fast because messages are small—72 bits. (Higher level protocols can obviously be layered on top of this, increasing the message size at the cost of speed.) Messages can be one of four primitive types, and the makeup of the message—the meaning of the 72 bits—depends upon this type. The types are **nearest neighbour** (NN), **point-to-point** (P2P), **multicast** and **fixed route** (FR).

TABLE 1
Internal Message Structure (Bitwidths in ())

Packet type	Data word (32)			Payload word (32)
NN	user			user
P2P	src node (16)	tgt node (16)		user
MC	src node (16)	src core (4)	src dev (12)	user
FR	user			user

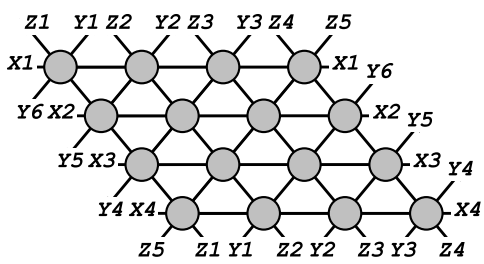


Fig. 3. The SpiNNaker interconnect topology.

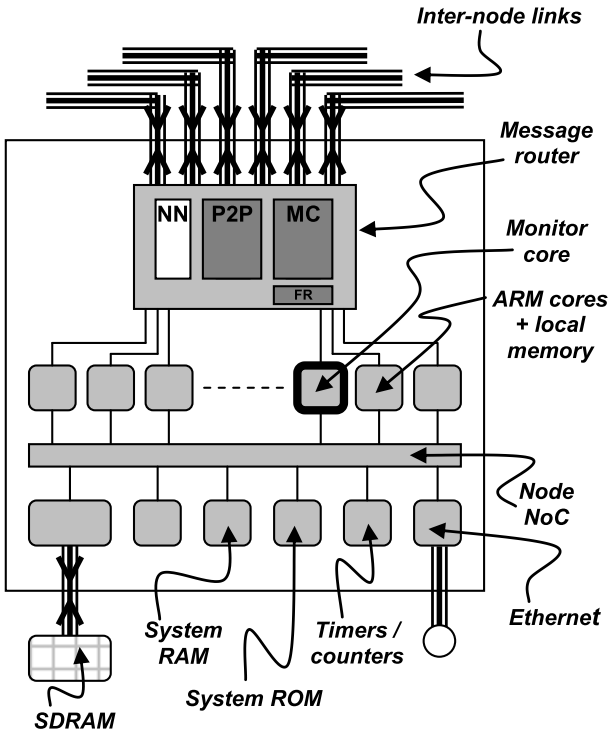


Fig. 4. Internal structure of a SpiNNaker node.

4.3 Resource Addresses

SpiNNaker is designed for the simulation of large systems that can be modelled as networks of small discrete entities, communicating with each other via small packets of information. The flagship application—neural simulation—gives the system its name: *Spi*king *N*eural *N*etwork *Archi*tect*ure*. The speed and scale of the system owes much to the fact that a lot of the infrastructure is hardware, rather than software. Consequently, the freedom usually enjoyed in labelling entities in software systems does not exist here. *Everything* is an unsigned integer, which allows us to pack information efficiently into messages.

Each application core can handle a number of entities (devices), the limit realistically being given by the size of the state space of each device and the physical memory available to the core. Within a 32-bit address space, we allocate 16 bits for the node address **(node ID)** and 4 bits for the core **(core ID)**, which leaves 12 bits for each device hosted by a core **(deviceID)**, making it feasible for the system to uniquely address 4,096 devices per core. The natural limit to the overall size of systems that can be *simulated* on SpiNNaker is over a billion devices.

4.4 Messages

Messages consist of a control byte, a data word, and an (optional) payload word—see Table 1. (Strictly, the payload being optional means that a packet size may be 72 or 40 bits, but in practice, the payload is almost always used, so it is easier to think in terms of a 72-bit packet.) The control byte contains the **packet type** (2 bits) and a variety of housekeeping data [1]. The type dictates how the packet is handled by the routing infrastructure, and (part of) the bit layout within the data word (which contains routing information used by the router hardware). The cells in Table 1 labelled 'user' are unused by SpiNNaker—the application programmer may use these bits.

4.4.1 Nearest Neighbour Messages

A NN message may be launched *from any core* (although in preferred usage it will only ever be the monitor), into a *set of output ports* (chosen by the generating core), whence it is delivered to the monitor core on the appropriate adjacent node. The generating core controls the content of the data word and payload, and once despatched, the message will be delivered to the monitor core of whatever node (or nodes) are physically connected to the chosen output ports - see Fig. 5. Thus the route of a NN message is fixed by the

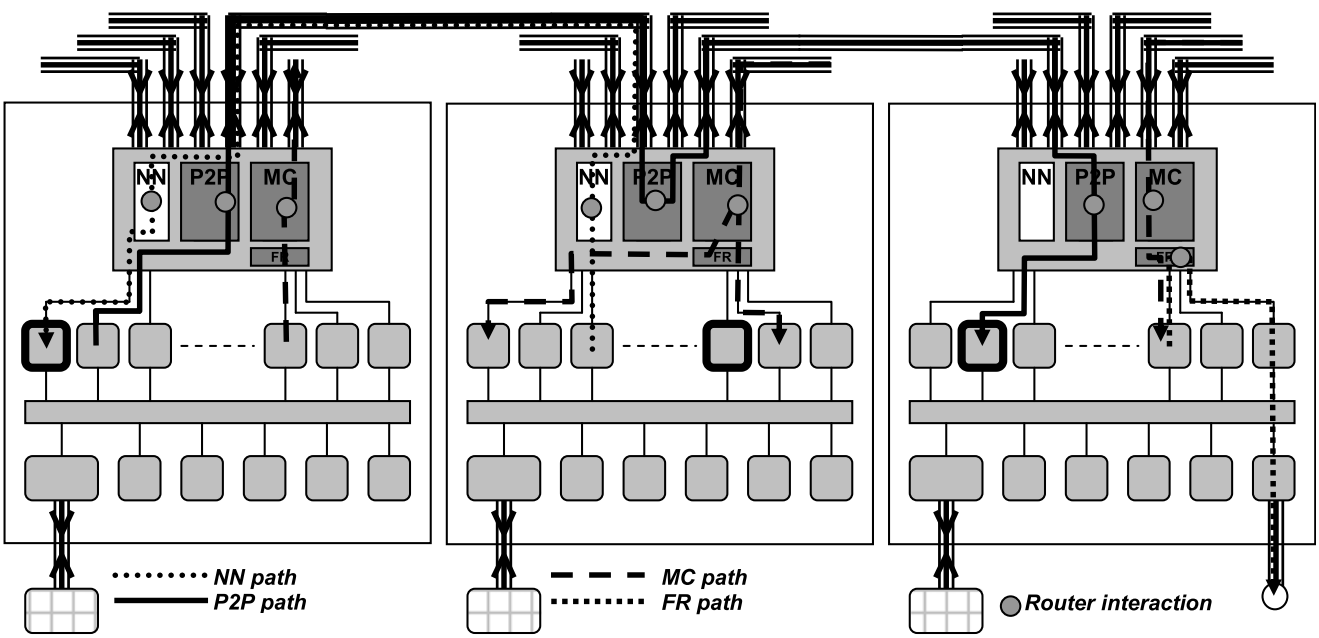


Fig. 5. SpiNNaker message types; router interactions.

hardware configuration—see Fig. 3—and requires no initialisation.

4.4.2 Point-to-Point Messages

A P2P message may be launched *from any* core, and will be delivered *to the monitor core* of the addressed target node. The generating core has (and needs) no knowledge of the route taken by the message—see Fig. 5.

Within the router of each node is a **P2P table**. It contains a 1:1 map of (target *node* address) => (output port). On any node, the router extracts the target node field from a P2P message (see Table 1), looks up the corresponding output port and forwards the message accordingly (except when the message has arrived at the target node, in which case it is forwarded to the local monitor core). The table *should* be complete, but if a P2P message is processed that has an unrecognised target address it will be dropped, and an error interrupt [1] sent to the node monitor. This illustrates an aspect of the design philosophy that is worthy of labouring: at every level of abstraction, wherever possible, the machine makes no assumptions about the integrity of its internal state. It should not be possible for a P2P packet to contain an address that has no match in a P2P table; but if it does, the system has a defined (and useful) behaviour.

Aside from the *initialisation* of the P2P tables in each router, the process is entirely hardware brokered. The P2P tables define a node topology which must be a function of the *working* processor mesh (that is, the subset of the system that is fault-free).

4.4.3 Multicast Messages

An MC message is (intended to be) used for device-level communication within a simulation. It may be launched by *any application core*, and will be delivered to a *set of target application cores* (which may be one)—see Fig. 5. The system makes use of a labelling methodology known as address event representation (AER) [17], taken from the world of neural simulation.

Whereas the NN and P2P messages are primarily used for initialisation and housekeeping functions, the MC packet is the 'simulation workhorse' packet. Although physically it is launched from an application core and delivered to an application core, in intended use it is more sharply focussed: it will be generated by an interrupt handler operating on a *device* (part of the problem graph) in one core, and delivered to a *device* in another core. The full address of every device modelled in a simulation is **node(16 bits):core(4 bits):device(12 bits)**—see Section 4.3. Each MC packet carries embedded within it this information for the *launching* device (Table 1) and the topology of the problem graph—embodied and distributed in the **MC route tables** of the system—ensures that the packet is delivered to the intended core(s). As part of the system initialisation process, a table is created in each node-local memory, defining the location of the target device state information, using the source device ID contained in the packet (Table 1) as a key.

The MC table is a complex (hardware) subsystem, consisting primarily of a content-addressable memory, described in [1]. It contains a 1:many map of (source device) => ({output port}, {target local application core}). If the entries in the table for the output port set or target local application core set are multi-valued, the router will "duplicate" the message and forward each copy. If the table contains no entry for an MC packet, it will simply be routed straight through the node, emerging from the (geometrically) opposite port to the one that it entered. This is the single point in the routing infrastructure design where the behaviour is based on the geometric, rather than topological attributes of the system, but the utility of the behaviour far outweighs its inelegance. Aside from the *initialisation* of the MC tables in each router, the process is entirely hardware brokered.

The MC tables effectively contain a distributed representation of the problem graph. The entries are thus Va function of the problem graph (which dictates which *device* is connected to which) and the P2P tables (which define how a message might get between specific *nodes*).

4.4.4 Fixed Route Messages

These are intended as a straight-through communication channel with the outside world. As with the other message types, their passage across the computing mesh is hardware brokered. They may be launched *from any core*, and will be delivered to the monitor core on the topologically closest node that has a connected Ethernet capability. (Internally, this is realised as a single entry MC table that matches every FR message.)

5 Bootstrapping

When the machine is powered up, virtually the only facility available is the NN packet routing, which is pure hardware and has no internal route tables that require initialisation. In this section, we describe the sequence of events necessary to initialise the SpiNNaker engine to the point where the simulation of a meaningful problem graph may be undertaken.

5.1 Initialisation

In order to perform useful calculations, the system needs to be initialised. Fig. 6 shows the interaction between the SpiNNaker system and its external software support. The vertical dividing line in the middle of Fig. 6 separates SpiNNaker internals from the outside world.

Externally, three tools are necessary, the **Loader**, the **Uploader** and a cross-compiler. The first two are bespoke; for the third, any commercial tool is suitable.

5.1.1 The Loader

Input to the Loader is

- The known topology of the processor mesh (including any known fault map)—i.e. what we know we have.
- The problem graph—i.e. the graphical description of the input problem—is described further in Section 6.

Output from the Loader is

- The contents of the P2P tables on each node (this is a function of the processor topology + fault map alone—it is independent of the problem graph).

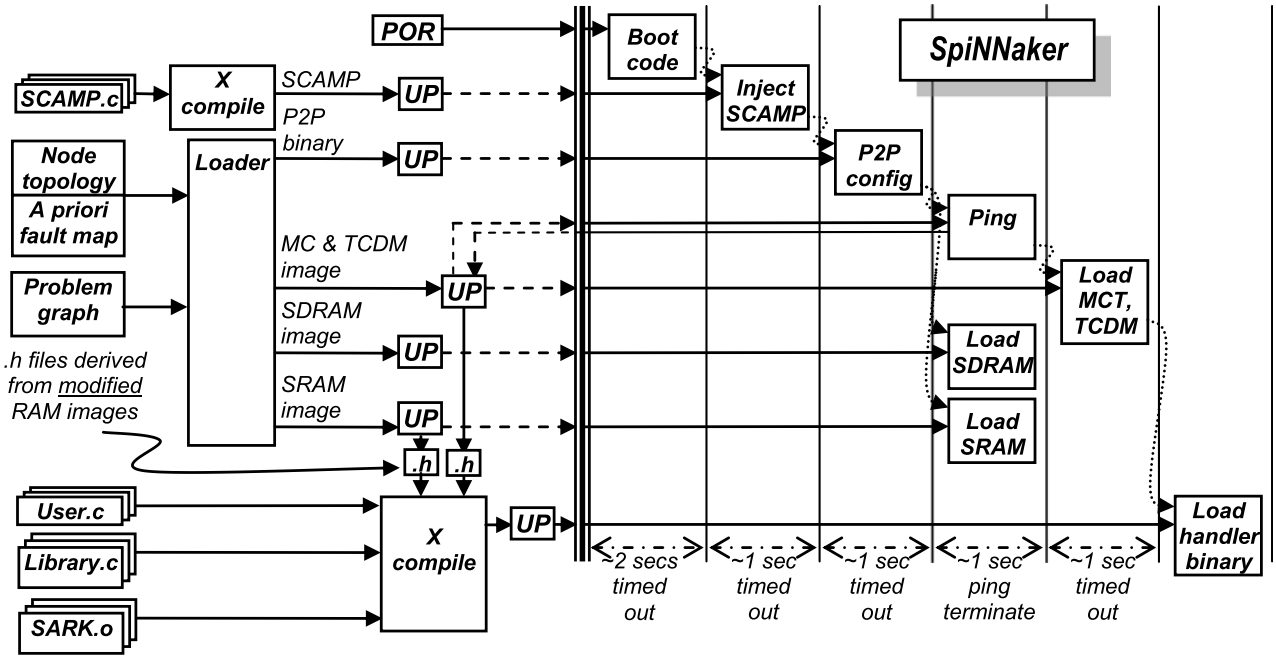


Fig. 6. Initialisation sequence.

- The contents of the MC tables for each node (these are functions of the P2P tables and the problem graph—Section 6 contains an example).
- The contents of various lookup tables that have to be written into the SDRAM and SRAM of each node, and the DTCM of each core.

5.1.2 The Uploader

The Uploader is shown in Fig. 6 as several small blocks (UP), to emphasize that the operations carried out are independent, although they are all embodied as one software tool.

Input to the Uploader are

- The various binary files generated by the Loader—solid lines.
- Some signals *from* the SpiNNaker engine via Ethernet—dotted lines. Throughout the rest of the paper, the Ethernet-connected node is referred to as the **root node**.

Output from the Uploader are

- Output files (the ***.h*** files shown in the figure)
- Control signals to the SpiNNaker engine itself (connected via the Ethernet port—dotted lines).

5.1.3 Cross-Compiler

Our language of choice for both the software development infrastructure and the code for SpiNNaker itself is C/C++; the user is expected to supply the source for the interrupt handlers in C, and the Uploader generates C header files. Consequently the cross-compiler must generate ARM binary from C. However, none of this is embedded into the design; virtually any (sensible) high-level language can be used.

The final component in the left side of Fig. 6 is the **POR** (power-on reset) signal, physically implemented as a push-button.

The right side of Fig. 6 shows the actions occurring inside SpiNNaker, arranged as a timing chart. The solid lines show information flow, and the predicate relationships are shown by the dotted curved lines. Thus, for example, the P2P configuration must terminate before the 'ping' process starts, but the SDRAM and SRAM loads may occur in any order, or, indeed, simultaneously.

1. *Boot code.* The POR causes the contents of the BOOT ROM to be copied into the ITCM of all the cores in all the nodes, and executed. In each node, these executing images perform a self-test, and working cores then take part in an (intentional) race, communicating via SRAM, to assign local identifiers (core IDs) to themselves. Thus one core will be elected the monitor core (ID:0) and up sixteen others allocated IDs 1..16. (In a perfectly functional node, then, one core will be unused). This mechanism allows nodes with less than 100 percent functionality to be useful. All the cores in a node are electrically equivalent; the nomination of one as monitor is (electrically) arbitrary. This process takes around 2 seconds, and is independent of machine size, because all the nodes boot simultaneously. There is no way for SpiNNaker to know when all its nodes have booted (cleanly or otherwise) so the process is timed out by the Uploader after 2 seconds.
2. *Inject SCAMP.* (***S*piNNaker *C*ontrol *A*nd *M*onitor *P*rogram) SCAMP is a control program (about 15 k binary) which is injected by the Uploader (via Ethernet) and loaded into the ITCM of the monitor core on the root node. It then copies itself into the ITCM of *all* the cores in *all* the nodes. (This is achieved by a combination of writing to shared memory—

SDRAM—to perform intra-node copies, and using ~3,750 NN packets to perform an inter-node flood-fill over the entire system.) The overall process is a self-timed pipeline, and the completion time is a function of the system size. As a reference point, it takes around 1 second on an 864 core system; quantitative timing data is available in [4]. As with the previous step, it is not possible for any one point in the system to know when the overall process has terminated, so this step is also timed out by the Uploader. At the end of this step, then, SCAMP is resident in the ITCM of *every* core in *every* node.

3. *P2P configuration.* At this point, it becomes possible to configure the data in the P2P routing tables, and assign system-wide unique identifiers to each of the nodes. This can be done in a number of ways. If the node topology is regular (the design intention) the P2P tables can 'self-organise': the root node allocates itself a compound identifier (0,0), and sets out a set of tokens (embodied as NN packets) to its nearest neighbours. Using knowledge of the incoming port and the generating node ID—enclosed in the 'user' fields of the packet—the receiving node can fill in a single entry in its P2P table. Subsequently, it does two things: it passes the token on to its nearest neighbours, to complete the search for the original node, and also it initiates a search wavefront for itself, enabling the system to populate further fragments of the P2P tables. In this way, the complete P2P table in each node can be assembled. The algorithm is simplistic, inasmuch as it makes assumptions about the node geometry, and is described in full in [4].

   If the node topology is not regular (as may be the case if a non-empty fault map exists), more sophisticated processing is required. A variant on the above can be used to populate the tables of arbitrary node topologies (this will be described in a later publication) or the data can be generated in the Loader, and injected into the system by the Uploader.

   In either case, like the previous steps, it is not possible to determine automatically when the process has terminated, so the Uploader times the step out after 2 seconds on the 864 core system.
4. *Ping response.* The Uploader interrogates each node in turn, to establish how many cores each has identified as functional. (This provides a rudimentary dynamic fault-mapping capability.) This information is used in the next few initialisation steps. It is gathered by the root monitor sending req/ack signals to every core in the system, via P2P packets to the monitor cores and shared memory (SRAM) messages to the consequent application cores. This step takes around 1 msec/core, the total time being roughly proportional to the system size.
5. *Load MC/TCDM.* The Uploader takes the images of the MC tables and the TCDM memory fragments, and uploads them. The information is *targeted* (it is different for each node) and transmitted to the recipient node by P2P packets. The content of the MC tables is described in Section 4, and the TCDM memory fragments are core-local tables that allow the interrupt handlers to locate data in the node-local memory.

   The information is generated by the Loader, based upon the node topology and any a priori faults supplied to it. However, if the ping response data garnered in the previous step shows that cores have gone out of service unknown to the a priori map, the Uploader can—up to a (small) point—modify the core assignment, by reworking the MC tables and TCDM maps such that references to the now faulty core are replaced by references to the 'spare' core on a node. Obviously this is only a viable tactic if a node has a core to spare—if any ping responses show > 1 unexpected cores at fault in a node, the entire initialisation has to abort.

   The information is also embedded in a machine-generated C header file that is cross-compiled with the user-supplied interrupt handlers. It contains the offsets for various Loader-generated lookup tables and the dynamic fault map derived in the previous section.
6. *Load SDRAM.* The SDRAM contains the state of the devices in the problem graph. It is a targeted load (each node has different information) brokered by P2P packets. It is unaffected by any core re-assignment and (almost) independent of machine size, but *is* a function of problem graph size.
7. *Load SRAM.* In this step, the SRAM tables are loaded (these are independent of the dynamic fault map), and another C header file generated. Again, this is a targeted load, and takes around 1 second (dependent on machine size) on an 864-core machine.

   (Loading the MC tables, TCDM, SRAM and SDRAM is a *node-by-node* targeted load so the Uploader knows when it has completed—there is no global timeout. Individual packet timeouts are used.)
8. *Load handler library.* Finally, the user binary is loaded. This binary is created externally by the cross-compiler, and is derived from a number of constituents:
   - 8.1: The header files generated by the Uploader—these reflect the dynamic fault map, and contain code offset data handed out of the Loader.
   - 8.2: The object code of SARK (***SpiNNaker a***pplication ***r***un-time ***k***ernel)—a static module that supports system-wide inter-processor communication and communication with the outside world via the root node.
   - 8.3: Library code (needs to be compiled with the two machine generated headers).
   - 8.4: The user code itself, describing the behaviour of the devices in the problem graph.

This binary image is written into a part of the ITCM that is unused by SCAMP; the final act of SCAMP is to transfer program control to SARK (and hence the user code). There is no return; although SCAMP still physically exists in each core, it is effectively orphaned at this point and becomes invisible.

At this point, SCAMP is effectively controlling the *monitor* cores, and a software stack SARK-library-user controlling all the *application* cores.

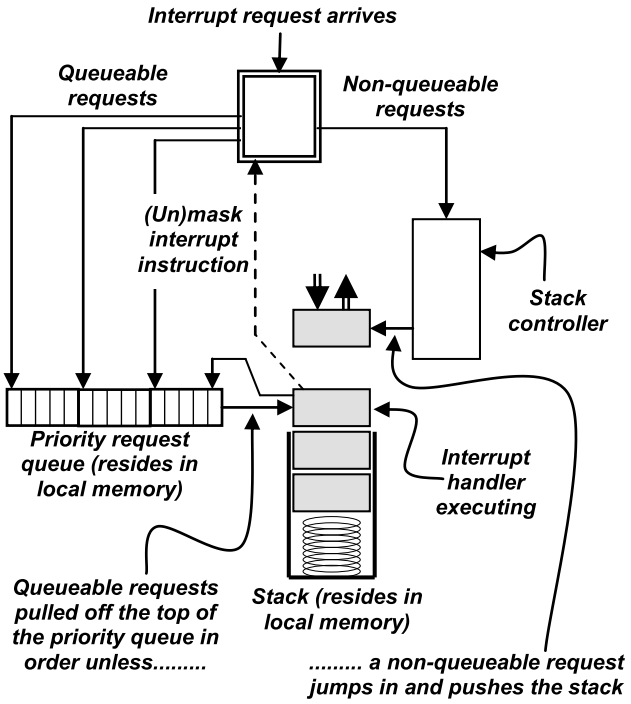


Fig. 7. Event handling.

5.2 Application Programming Model—SARK

The SpiNNaker programming model is a simple, **event-driven model**. *Applications do not control execution flow*, they only indicate the functions (event handlers), to be executed when specific events occur, such as the arrival of a packet, a software-generated interrupt from an application core or the lapse of a periodic time interval. SARK controls the flow of execution and schedules the invocation of the handlers.

Fig. 7 shows the architecture of the event-driven framework. Application developers write event handler routines that are associated with events of interest and register them at a certain priority with the kernel. When the corresponding event occurs the scheduler either executes the handler immediately and atomically (in the case of a non-queueable handler) or places it into a scheduling queue at a position according to its priority (in case of a queueable handler). When control is returned to the dispatcher (following the completion of a handler) the highest-priority queueable handler is executed. Queueable handlers do not necessarily execute atomically: they may be pre-empted by non-queueable handlers if a corresponding event occurs during their execution. The dispatcher goes to sleep (low-power consumption state) if the handler queue is empty but will be awakened by any subsequent event.

The SpiNNaker application programming interface (API) supports the programming model providing functions to register handlers, enter and exit critical sections, communicate with other cores and the host, trigger DMA operations and other useful tasks. In all, 32 different types of interrupt are supported—these are detailed in [1].

6 Computing with Interrupts

6.1 Interrupt Handler Programming Environment

In a conventional parallel system the user can reasonably expect a comprehensive computing environment and an application programming interface to be provided. This will include file and console input/output (I/O), memory management (a heap manager for dynamic memory allocation), software libraries (including some message passing infrastructure), and some notion of temporal coherency and the passing of real time.

In SpiNNaker, almost none of these are available. Each packet interrupt handler has read access to the bits of the packet that triggered it; knowledge of the local physical port by which the packet arrived, I/O to its own memory map; knowledge of its own core ID (0..16) and node ID ($0..2^{16}$); the ability to launch packets, and a coarse (ms) timer (which has an associated interrupt, for which the user can provide a handler). The interrupt handlers are an ensemble of (necessarily small) program threads, each invoked by the hardware in response to a specific incoming hardware event—the arrival of an interrupt.

- There is no direct file or console I/O from a core: the sheer size plus the isotropic and homogeneous nature of the architecture of the system precludes this. Design provision is made for each *node* to connect to the outside world, but in practice we communicate via a single link (Fig. 5) and a set of handlers in each core that allow the transient creation of communication channels between any core and the outside world, which is a cumbersome process.
- There is no memory management: Although each core has a full 32 bit memory map, it has only 64k DTCM and 32 k ITCM. There is little room for a memory manager, and the design intention is that the individual handler threads are very simple—handlers *requiring* internal memory management are way outside the design spirit and intention of the architecture.
- There is no interactive debug, because there is no notion of an overseer process or temporal coherency across nodes. SpiNNaker is designed to simulate systems in which *time models itself*—the devices of the problem graph *asynchronously* communicate amongst themselves. The user *could* inject 'pause', 'read' and even 'write' command packets into the system, but would have no control over when they might arrive, or what state the machine might be in when they do.
- There is no MPI-type message passing system. The memory footprint is too big, the resources to support it do not exist, and the physical limitations on the SpiNNaker packet size (and hence bandwidth for large messages) would make the system unusable.
- The physical difficulty of providing a rigorous temporal synchronisation capability led us to discard this very early on. A coarse (O(ms)) timer interrupt provides rough knowledge of the passing of wall-clock time.

6.2 Algorithmic Concerns—Neural Simulation

The flagship application—for which the hardware is optimised—is neural simulation. At the level of granularity at which we consider matters, neural systems are composed of **neurons**, that communicate via **action potentials** (spikes) that travel between the neurons along **axons**, terminating at

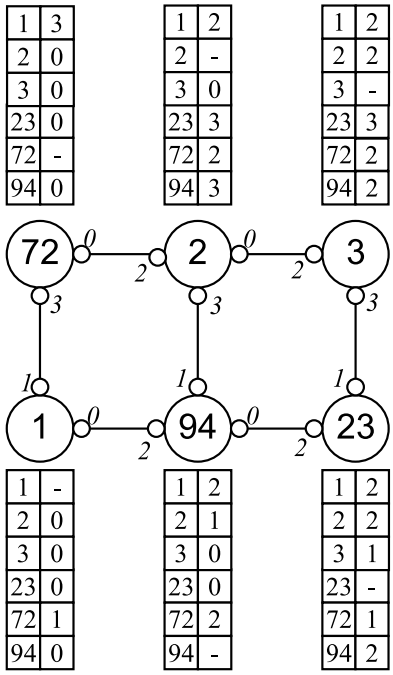


Fig. 8. Cut-down SpiNNaker processor mesh.

a **synapse** on the target neuron. It is (almost) a discrete system, but with terminology that may be alien to an engineering audience. The problem is, then, that of discrete simulation, and the underlying hardware turns it into the *parallel discrete simulation problem*, which has been the subject of attention for decades [18], [19]. What sets SpiNNaker apart in this application is the manner in which simulation causality is handled. In a conventional parallel simulation system, non-trivial effort is required to maintain simulation causality across the computing ensemble. SpiNNaker avoids this computational overhead by simply ignoring it. Biological neurons (all) operate at frequencies of up to around a kilohertz, and neural signals propagate at speeds of a few ms^{-1}. This means that the propagation delay of packet traffic throughout the compute fabric is completely negligible compared to the biological delays intrinsic to the system being simulated. *Biological* delays are modelled by local real-time physical delays implemented on the ARM cores, and time effectively models itself: events arrive "infinitely fast", are delayed by a biologically realistic amount, then processed "infinitely quickly" and any consequent events immediately broadcast.

These modelling compromises enable the cores to operate at full performance, giving each node (with 18 200 MHz cores) approximately the same compute performance on this task as an Intel ATOM N270[2] processor, but with a power budget of 1 W.

The prototype development flow has so far been used to develop small models (up to a few 10,000 s of neurons) [16].

2. The Intel Atom N270 single core processor delivers ~3.8 GIPS at 1.6 GHz; Spinnaker with 17 cores at 200 MHz delivers ~4 GIPS. Both these figures are peak performance, and both will be adversely affected by poor data locality. On Spinnaker, the memory hierarchy is organised to ensure near-perfect data locality on suitable (small) problem fragments. This is much harder to organise (and impossible to guarantee) on a cached processor such as Atom.

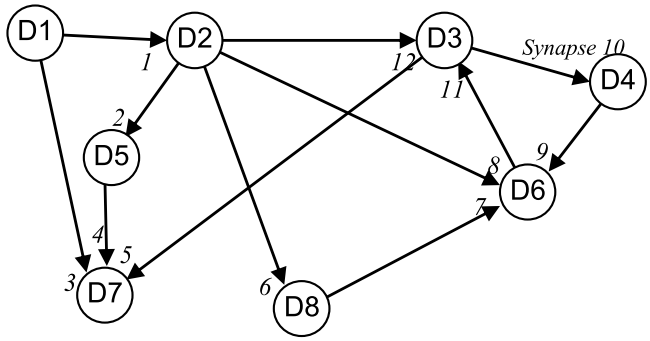


Fig. 9. Example problem graph.

Similar models have been demonstrated in real-time robotics control [2] and simple vision applications.

SpiNNaker is designed as a computing engine that performs by the asynchronous exchange of many small packets. A useful way of thinking about the system is to view it as a large, distributed finite state machine or Petri net. The fragment of the overall state embodied by a specific node/core *may* be changed by an interrupt handler triggered by an impinging packet. (This view is valid for almost any computing engine, of course, but it is particularly useful in the case of SpiNNaker.)

6.3 Simulation of a Simple Example

Here, we present a reasonably detailed example of how a very simple problem graph might be loaded onto a very simple, cut-down SpiNNaker engine, and how one might perform a meaningful simulation within the architectural constraints of SpiNNaker.

We do *not* describe:

- *How* the P2P tables are initialised.
- *How* the problem graph is mapped onto the SpiNNaker core graph.
- *How* the MC tables are generated.

- we simply present the information here.

Fig. 8 shows a node-level representation of a much reduced SpiNNaker system, consisting of six nodes (72, 2, 3, 1, 94, 23) connected as shown. The nodes are interconnected by just seven links; and the ports—where present—labelled 0..3. The P2P tables associated with each node are also shown. Fig. 9 shows a directed problem graph. The devices (D1...D8) drive each other via labelled (1...12) connections. Fig. 10 shows a possible mapping between problem graph and node graph, and Fig. 11 shows the corresponding MC table entries for the system. Also in the nodes of Fig. 11 are the node-local device lookup tables (**DLT**s).

For the rest of the example, we will use the term **neuron** for problem devices, and **synapse** for connection. For the sake of illustration, assume a handler in node 72, core 14 (72 | 14) emits a packet (spike) from D1. This is realised as an MC packet, Fig. 9 shows us that this must be delivered to D2 and D7. How do the data structures of Fig. 11 support this?

The MC packet generated by core 14 is transmitted to the multicast route table (Fig. 5) in node 72. The data word in the MC packet (Table 1) is 72:14:D1. The router in node 72 will use 72:14 as a key for the MC table, and finds that the packet is to be sent to both port 0 and core 15. The packet

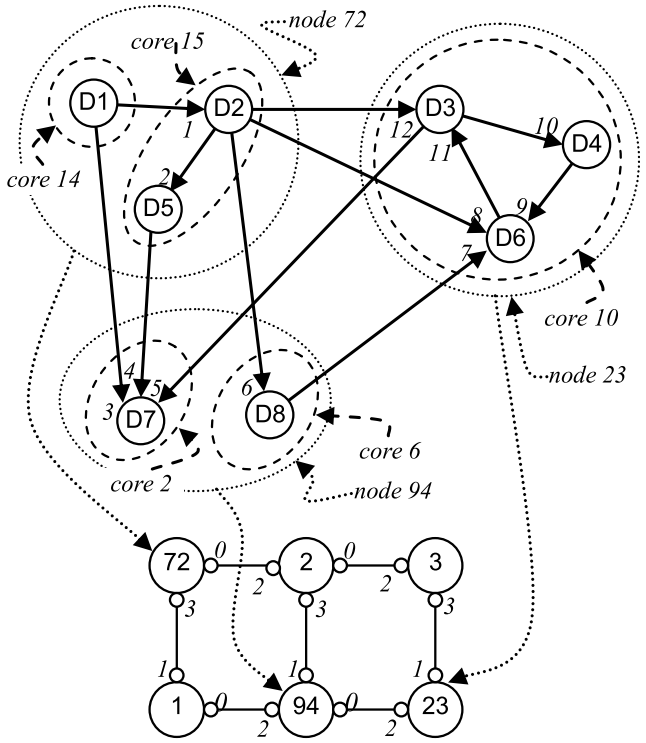


Fig. 10. Mapping of Fig. 9 into Fig. 8.

data is duplicated (recall this is hardware) and the two copies launched.

The copy arriving at core 15: will cause an interrupt. The handler will read the packet data (72:14:D1), and from the node-local connectivity table (DLT)—Fig. 11—see that if the *source* neuron is D1, the *target* neuron must be D2 *on this node/core*. (If the table has no entry, the packet is simply dropped.) The handler then modifies the *state* of D2 (which may or may not cause subsequent packets to be generated), and *terminates*.

The copy sent from port 0: arrives (at port 2) of node 2 (Fig. 11). 72:14 matches the entry in the MC table on node 2 (retrieving port 3, no cores) and the data is forwarded out of node 2 via port 3. This arrives at port 1 on node 94, and matches the entry in the MC table in node 94, retrieving no ports, core 2. The handler on core 2 is triggered; the packet data shows the *generating* device to be D1, and the node-local connectivity table shows the target neuron to be D7, and the handler may modify the state of D7 and/or launch packets (from D7).

From the perspective of biology, SpiNNaker is *fast*. Node-node packet transit time is ∼100 ns, and the design intention is that the handlers should be comparable in speed. The real-time clock interrupt enables interrupt handlers to keep track of 'real time', and *delay* the emission of generated packets to biologically realistic times.

The above explanation charted the movement of one packet across the SpiNNaker fabric, but the system is massively parallel: in principle, there can be millions of packets 'in flight' simultaneously.

6.4 Event Handlers

The final component of the system necessary are the *event handlers*. The previous sections of the paper have been domain-agnostic; we have described the functioning of SpiNNaker in abstract terms, and these remain valid for every application domain. The event handlers embody the behaviour of the problem devices and the interpretation of messages.

For the sake of explanation, let us consider an extremely simple device (this would be, for example, one of the nodes in Fig. 9): a leaky integrate-and-fire pulse generator. The packets passed between devices represent pulses (which we will assume for the sake of simplicity to have unity weight). On receipt of a packet, a device will increment an internal counter (the *state*—this supports the 'integrate' behavioural component). When a certain threshold is reached, the device

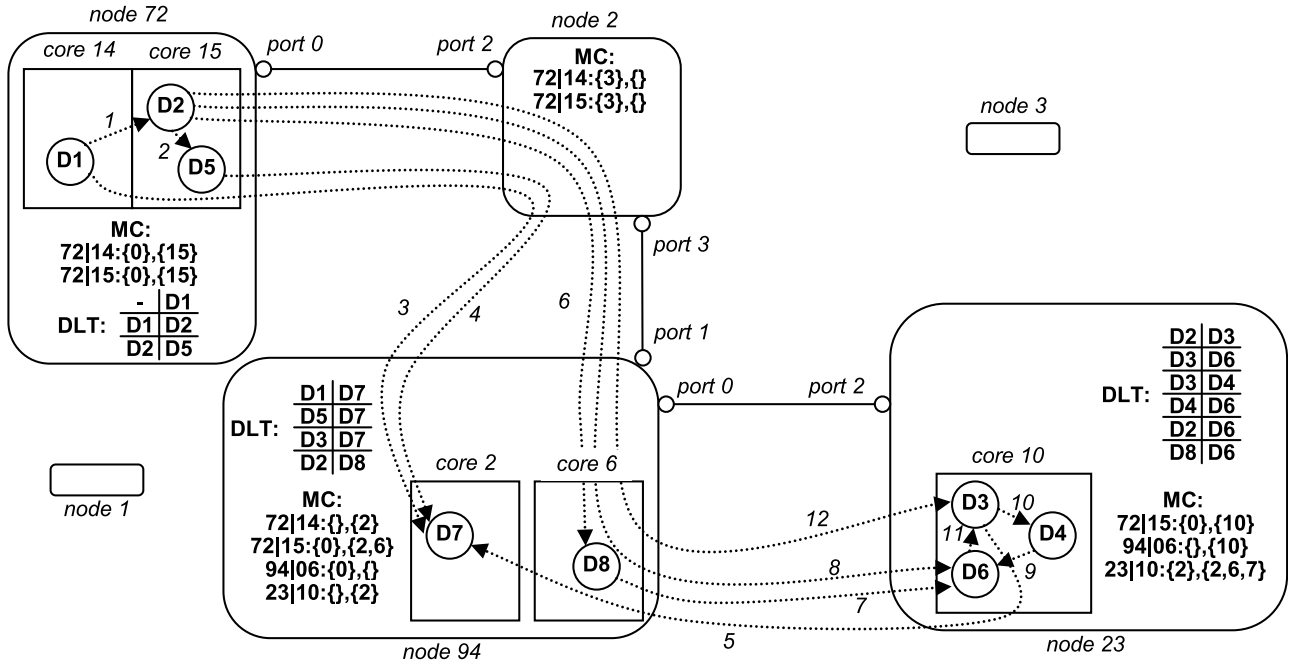


Fig. 11. Node datastructures.

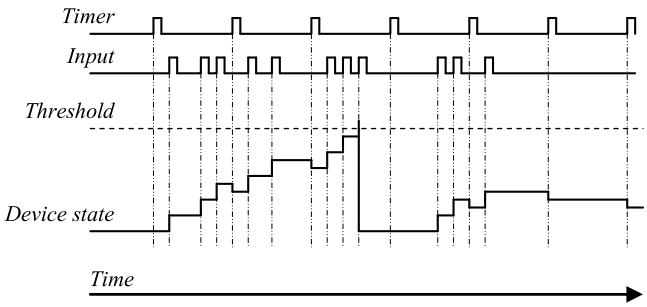


Fig. 12. Simulated behaviour of the LIF device.

emits a packet of its own, and resets its state to 0. Alongside this, *every* device is regularly informed of the passing of wallclock time by timer packets delivered to it by the *hardware*. On receipt of this packet, the internal state value of each device is reduced by, say, 95 percent—(this supports the 'leaky' behavioural component).

The device behaviour outlined represents an extremely simplistic neuron model, but it is not hard to see how this could be re-interpreted in different physical domains.

The user must now provide two event handler functions, one (triggered by the hardware on arrival of a pulse packet), the other on arrival of a wall-clock tick:

```
void OnPulse(MC_t *)
{
if (++state<=threshold)
  return;
SendPulse();
state = 0;
}
```

```
void OnTimer(TT_t)
{
state *= 0.95;
}
```

The incoming packet contents are available to `OnPulse`, and the time to `OnTimer`, via their arguments, but are not used here. The *existence* of the interrupt carries sufficient information for the computation.

```
void OnPulse(MC_t * packet)
{
switch (packet->payload.type) {
  case PULSE : if (++state<=threshold)return;
               SendPulse();
               state = 0;
               return;
  case THRESH: threshold = packet->payload.th;
               return;
  case RESET : state=packet->payload.S0;
               return;
  case OUTPUT: oflag = !oflag;
               return;
}
}
```

```
void OnTimer(TT_t time)
{
state *= 0.95;
if (oflag) SendPulse(BuildPkt(time));
}
```

The behaviour of a single device, subject to an incident pulse train, is shown in Fig. 12.

It is not hard to see how a much richer set of behaviours may be supported with the above framework:

Returning to Fig. 2, we have now outlined all the information necessary for both arms of the dataflow shown: the machine topology (Fig. 8), the problem graph (Fig. 9) and the individual device handlers described above.

6.5 Output

Output from the system can take a number of forms, depending on the nature of the computation.

- When performing neural simulation, the system is designed to operate in real time. Specific devices are inserted into the problems graph, known as **monitor devices** (not to be confused with monitor cores). These do not represent physical entities; rather they host a different set of event handlers. When a MC packet is incident on a monitor device, the handler wraps the data in a higher level protocol and redirects it to the monitor core on a node connected to the Ethernet, and hence to the outside world. This may be done using FR, P2P or MC packets (see Section 4). Two further points are relevant: (1) the monitor device may buffer the incident packets and forward them as a bundle, if the timing information intrinsic to the absolute packet arrival time is not compromised; (2) each SpiNNaker *node* contains an Ethernet controller—an arbitrary number of these may be physically connected to the outside world.

Alternatively, simulator results may be written to the SDRAM on each node, and harvested and transmitted to the outside world by a post-simulation program (a reaper) run after the main simulation is over.

6.6 Application Portfolio

SpiNNaker is a massively-parallel packet-mediated simulation engine, and its position in the packet size/cost spectrum (Fig. 1) makes it ideally suited for *certain types* of simulation. The attribute that these simulation types have in common is the absence of a central computational overseer. The impact of any such overseer has a dramatic effect on the computational throughput; SpiNNaker is intended for situations where the many, small, interacting cores can behave autonomously.

The types of simulation for which SpiNNaker is ideal fall roughly into two classes, mimicking the output strategies outlined in the previous section.

- *Event-brokered systems*. Neural simulation, discrete system simulation, some representations of molecular dynamics. The problem is perturbed into a form whereby locally autonomous devices react independently via packets delivered through a network, the topology of which is complex and an integral part of the system under simulation (neurons, electronic circuits).
- *Relaxation-based systems*. Finite difference (diffusion), some representations of molecular dynamics/computational chemistry, large matrix mathematics. The problem is transformed again into a set of devices, but here the connection topology is derived closely from the *geometric* relationships of the devices.

7 Future Challenges

SpiNNaker is a massively ambitious undertaking, which has been almost a decade in gestation. It has now got to the state of beginning to deliver quantitative results, and it is performing—so far—almost exactly to expectations. However, a host of problems remain unsolved:

7.1 Inline Place and Route

When SpiNNaker reaches its target size—2^{16} nodes—it will be capable of simulating systems of 10^9 devices. The routing tables, at least, are of fixed size, and in total occupy around 1.4 Gbyte. The total device state memory footprint—assuming it is resident in the SDRAM—is limited to just over 7 Tbyte. The offline manipulation of this quantity of data, let alone the upload task, is a ferocious challenge, and naturally one looks for ways around—rather than through—the problem.

One obvious technique is to present SpiNNaker with the topological connectivity of the problem graph—or even some high-level representation of it—and persuade SpiNNaker to generate the internal data structures itself. Whilst we have some preliminary ideas [3], this in itself will probably require several years of effort.

7.2 Real Time Route Table Reconfiguration

The problem of dynamic changes in the topology of the problem graph is a characteristic of real, neurological systems. The problem is at least *architecturally* localised: we need to be able to dynamically change the contents of the MC table. However, the table keys are aggressively compressed, and any attempt at modulating the route information embedded therein requires, at the very least, unpacking all or some of the tables on a specific route, both pre-and post reconfigure.

7.3 Real Time Fault Tolerance

Biological neural systems exhibit remarkable fault tolerance at the connectivity level, and our long-term ambitions for this project include both using massively parallel computing resources to accelerate our understanding of brain function, and utilising a growing understanding of brain function to point the way to more efficient parallel, fault-tolerant computation.

It is relatively simple to time-slice into the operation of the simulation a packet-mediated network searching algorithm that continuously monitors the health of the physical compute fabric—the subsequent modification of the routing tables (Section 5) is non-trivial. However, it is a necessary but unsatisfactory procedure: necessary, because hardware does fail in use, and we have to be able to cope with this; unsatisfactory, because it is an engineering solution that does not mimic biology.

8 Final Comments

SpiNNaker is a hugely complex system, and its development is pushing at a number of intellectual boundaries simultaneously: the hardware build (a million cores, communications infrastructure, power management, storage and manipulation of state data) and software development. The general case parallelisation problem is one of the outstanding unconquered holy grails of computer science—with SpiNNaker, there really is no other way of doing it, and one of the long-term objectives of the project is a general-purpose formalism for large-scale fine-grain parallel programming.

The system has two outstanding practical advantages:

- The circled area of Fig. 1 is where SpiNNaker wins in terms of message cost.
- A 'conventional' parallel supercomputer can cost of the order of GBP1-2 k per core; SpiNNaker (to manufacture) costs around GBP1 per core.

This paper has outlined the low-level programming techniques so far employed to underpin system development of this nature. Whilst far from a generic solution technique, general principles are beginning to emerge, and a way of thinking about the necessary problem formalism starting to crystallise.

Acknowledgments

This work was supported by the United Kingdom Engineering and Physical Sciences Research Council (under EPSRC grants EP/G015740/1 and EP/G015775/1), with industry partner ARM Ltd.

References

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

[2] S. Davies, C. Patterson, F. Galluppi, A. D. Rast, D. R. Lester, and S. B. Furber, "Interfacing real-time spiking I/O with the SpiNNaker neuromimetic architecture," in *Proc 17th Int. Conf. Neural Inf. Process.*, Sydney, Australia, 2010, pp. 7–11.

[3] A. D. Brown et al., "A communication infrastructure for a million processor machine," in *Proc. 7th ACM Int. Conf. Comput. Frontiers*, Bertinoro, Italy, May 2010, pp. 75–76.

[4] T. Sharp, C. Patterson, and S. B. Furber, "Distributed configuration of massively-parallel simulation on SpiNNaker neuromorphic hardware," in *Proc. Int. Joint Conf. Neural Netw.*, San Jose, CA, USA, Jul. 2011, pp. 1099–1105.

[5] D. E. Shaw et al., "Anton, a special-purpose machine for molecular dynamics simulation," *Commun. ACM*, vol. 51, no. 7, pp. 91–97, Jul. 2008.

[6] J. Howard et al., "A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, 2010, vol. 9, no. 2, pp. 108–109.

[7] D. Fick et al., "Centip3De: A 3930DMIPS/W configurable near-threshold 3D stacked system with 64 ARM Cortex-M3 cores," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, 2012, pp. 190–192.

[8] S. Satpathy et al., "A 4.5Tb/s 3.4Tb/s/W 64 $\times$ 64 switch fabric with self-updating least-recently-granted priority and quality-of-service arbitration in 45nm CMOS," in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, 2012, pp. 478–480.

[9] S. Bell et al., "TILE64TM processor: A 64-Core SoC with Mesh interconnect" in *Proc. IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, 2008, pp. 88–598.

[10] M. Hines, S. Kumar, and F. Schürmann, "Comparison of neuronal spike exchange methods on a Blue Gene/P supercomputer," *Frontiers Comput. Neurosci.*, vol. 5, p. 49, Jan. 2011.

[11] S. Choudhary, S. Sloan, S. Fok, A. Neckar, E. Trautmann, P. Gao, T. Stewart, C. Eliasmith, and K. Boahen, "Silicon neurons that compute," in *Proc. 22nd Int. Conf. Artif. Neural Netw. Mach. Learn.*, 2012, vol. 7552, pp. 121–128.

[12] S. Millner, A. Grübl, K. Meier, J. Schemmel, and M.-O. Schwartz, "A VLSI implementation of the adaptive exponential integrate-and-fire neuron model," in *Proc. Adv. Neural Inf. Process. Syst.*, 2010, vol. 23, pp. 1642–1650.

[13] N. Imam, P. Merolla, J. Arthur, F. Akopyan, R. Manohar, and D. Modha, "A digital neurosynaptic core using event-driven QDI circuits," in *Proc. IEEE 18th Int. Symp. Asynchronous Circuits Syst.*, Lyngby, Denmark, May 7–9, 2012, pp. 25–32.

[14] C. Mead, *Analog VLSI and Neural Systems*. Reading, MA, USA: Addison-Wesley, 1989.
[15] N. J. Boden et al., "Myrinet: A gigabit-per-second local area network," *IEEE Micro*, vol. 15, no. 1, pp. 29–36, Feb. 1995.
[16] T. Sharp, F. Galluppi, A. Rast, and S. B. Furber, "Power-efficient simulation of detailed cortical microcircuits on SpiNNaker," *J. Neurosci. Methods*, vol. 210, no. 1, pp. 110–118, Sep. 2012.
[17] M. Mahowald, *An Analog VLSI System for Stereoscopic Vision*. Norwell, MA, USA: Kluwer, 1994.
[18] R. M Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.
[19] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Trans. Softw. Eng.*, vol. SE-5, no. 5, pp. 440–452, Sep. 1979.

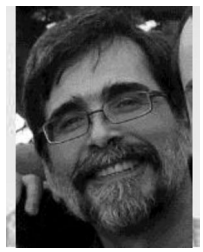
Andrew D. Brown (M'90-SM'96) is a professor of electronics at Southampton University, United Kingdom. He has held visiting posts at IBM Hursley Park, United Kingdom, Siemens Neu-Perlach, Germany, Multiple Access Communications, United Kingdom, LME Design Automation, United Kingdom, Trondheim University, Norway, and Cambridge University, United Kingdom. He has held a Royal Society industrial fellowship, and published more than 150 papers. He is a fellow of the IET and BCS, a chartered engineer, and a European engineer. He is a senior member of the IEEE.

Steve B. Furber (M'98-SM'02-F'05) is an ICL professor of computer engineering in the School of Computer Science at the University of Manchester. He was at Acorn Computers during the 1980s, where he led the development of the first ARM microprocessors. He is a fellow of the Royal Society, the Royal Academy of Engineering, the British Computer Society, the Institution of Engineering and Technology, and the IEEE.

Jeffrey S. Reeve (M'95-SM'01) received the PhD degree in theoretical physics from the University of Alberta, Canada, in 1976. He is a senior lecturer at Southampton University, United Kingdom. He was at the Communication and Control Group of Plessey, Auckland, NZ and the Airspace division of Marconi Radar, Chelmsford, United Kingdom. He has more than 100 publications in distributed computing, network security and management. He is a chartered physicist and a member of the IoP. He is a senior member of the IEEE.

Jim D. Garside received the BSc degree in physics in 1983 and the PhD degree in computer science in 1987, from the University of Manchester, United Kingdom. After a brief sojourn in the software industry, he returned to the University of Manchester as a lecturer in 1991. His current research interests include power-efficient processing especially using hardware reconfiguration.

Kier J. Dugan received the MEng degree in electronic engineering from the University of Southampton, where he is currently working toward the PhD degree in electronics and computer science. His research interests include self-configuration of distributed computing systems, high-performance networks, and computer architecture.



Luis A. Plana (M'97-SM'07) received the PhD degree in computer science from Columbia University. He is a research fellow in the School of Computer Science at the University of Manchester, United Kingdom. His research interests include the design and synthesis of asynchronous, embedded, and GALS systems. He is a senior member of the IEEE.

Steve Temple received the PhD degree in computer science from the University of Cambridge. He is a research fellow at the School of Computer Science at the University of Manchester. His research interests include self-timed logic, VLSI design, and microprocessor system design.