Debugging SpiNNaker programs on a computer by API emulation

Thomas Rast*

May 29, 2012

Abstract

The SpiNNaker hardware consists of many ARM9 cores running in parallel, connected by a custom communications fabric. Thanks to an event-driven API, it can be programmed in standard C. Debugging such programs is difficult, however. We show that the API can be emulated faithfully enough to run unmodified SpiNNaker programs on Linux, simply by compiling them for 32-bit x86 and linking them to the emulator. This makes the entire range of Linux debugging utilities available for use on SpiNNaker programs.

1 Introduction

The SpiNNaker architecture [PFT+07] designed by the project of the same name at the University of Manchester is a globally asynchronous system composed of ARM9 processor cores and hardware networking. Its ultimate goal is to assemble a network of SpiNNaker chips capable of simulating up to a billion neurons in real time. A brief description of the hardware is given in Section 1.1.

Despite being an ARM9 based architecture, the hardware limitations prevent software support for other ARM platforms (such as smartphones) from carrying over directly to the SpiNNaker. Software must be written specifically for the SpiNNaker chip. To that end, cross-compilers and assemblers running on computers are readily available.

From bottom (closer to hardware) to top, the software stack that has emerged for SpiNNaker has, roughly speaking, four components:

1. A hardwired bootloader.
2. The SpiNNaker Control and Management Program (SC&MP), loaded by the bootloader. It supports loading data and starting applications over Ethernet.
3. The Application Run-Time Kernel (ARK) and Application Programming Interface (API), which together provide an event-driven programming model for user applications [SPGF11].

*Thomas Rast is with the Institute of Theoretical Computer Science, ETH Zurich, Switzerland (email: `trast@inf.ethz.ch`).

4. A PyNN backend [RGD+11] – running on a computer – that translates PyNN neural network descriptions into data and application packages that run the corresponding networks on the SpiNNaker system.

For anything not directly supported by the PyNN backend, custom software must be written. This is usually done in C using the event-driven API in layer 3. The rest of this paper is concerned only with such C programs.

1.1 Summary of SpiNNaker hardware

We only give a brief summary of the points that are relevant to this paper. A full description of the hardware is given in [PFT+07].

Each SpiNNaker chip has the following components:

- 17 ARM9 processor cores. Each core is equipped with 64kB of data and 32kB of instruction tightly coupled memory (DTCM/ITCM). This memory is only accessible from the core itself.
- A 128MB SDRAM module, shared between all cores, accessible using DMA.
- Six bidirectional connections for message passing into/out of the chip.
- A hardware packet-switched router which uses a configurable routing table to route (single- and multicast) packets from/to the 6 external links and the processor cores on this chip.

A SpiNNaker system then consists of a number of SpiNNaker chips, assembled into a toroidal topology using the six available connections (four to form a grid, and two for the main diagonal). In addition, some chips may be equipped with an Ethernet interface to exchange data with computer systems.

All of this communication is unordered and lossy: packets may arrive in a different order than they were received, or may not arrive at all. High I/O load, high CPU load, and multicast storms (e.g., because of misconfigured routing tables) can cause buffers to fill, at which point the routers will drop packets until there is room. Also note that chips which do not have an Ethernet port must route all data from/to the Ethernet over a chip that does (the SC&MP software already takes care of this), further increasing I/O contention.

In the sample graciously provided to us by the SpiNNaker project, there are four chips in a 2×2 grid configuration. The chip at position $(0, 0)$ has an Ethernet interface. All our code is written with this topology in mind, but the methods generalize to an arbitrary number of chips.

1.2 Brief overview of the SpiNNaker API

The API used for SpiNNaker programming, described in [SPGF11], provides an event loop driven system. The user configures *callback functions* and priorities for each of the supported events, and invokes the main entry point of the *dispatcher* (called `spin1_start()`). The dispatcher then waits for events, queues them, and invokes callback functions on queued events in the order of their priority.

The API does not provide any management of the SDRAM. The programmer may instead use and access it at a fixed address, and optionally implement custom management, if any should be needed.

This provides a very familiar interface for most programmers: execution on a single processor core is strictly sequential, and event-driven frameworks are widely used (e.g., in GUI toolkits and webserver programming).

On the other hand, the workflow proves to be very inconvenient for programmers used to working in a *nix environment. Programs had to be run on the SpiNNaker itself because the platform is so different from an ordinary computer. As of this writing, there is no debugger or similar tool that runs on SpiNNaker. This makes the `printf()` family the debugging method of choice. Additionally, the SDRAM can be accessed via the SC&MP. However, as mentioned above, all of this communication is lossy and must pass through a core with an Ethernet interface, which sometimes makes it impossible to get any output from a program.

These problems motivated our work, which we discuss next.

1.3 Our contribution

We discuss a method of emulating parts of the SpiNNaker infrastructure, namely the memory layout and the SpiNNaker API, in a sufficiently faithful way that lets us run SpiNNaker programs on an ordinary computer. Our work was done on Linux, but uses standard POSIX interfaces, so that the same should work on any other system that conforms to POSIX.

[**TODO**: more]

2 Emulating the SpiNNaker API

Various levels of emulation are possible. At the lowest level, for example, the full state of the emulated system can be *simulated* on the host system. This is the approach taken, for example, by the QEMU project. However, this has many drawbacks, not least that it is a massive effort to implement and comparatively slow to execute. At the other end of the scale, software like Wine relies on substituting a new implementation of the underlying API and then running the program natively. We are taking the latter approach here.

As explained in Sections 1.1 and 1.2, there are three main ingredients to a sufficiently faithful emulation of SpiNNaker:

1. The hardware memory mappings.
2. The communications fabric.
3. The SpiNNaker API.

In the following we discuss how we implement each of these on an ordinary Linux system. Our software then acts like a substitute library against which a SpiNNaker program can be linked to let it run on Linux. For easier reference, we call the Linux system the *host*, the library the *emulator*, and the SpiNNaker program the *program*.

Table 1: Memory mappings in the SpiNNaker API

Memory	Base	Size
ITCM	0	32kB
DTCM	0x400000	64kB
SDRAM	0x70000000	128MB
SYSRAM	0xf5000000	16 or 32kB

Note that due to some design choices inherent in the API, C code written to it is not portable to 64-bit systems. It is therefore necessary to compile both the program and the emulator as 32-bit code.

2.1 Memory mappings

There is a number of memory mappings exposed to the program. The memory mappings relevant to this paper are listed in Table 1.

We mostly ignore the ITCM in this work. It is designed for instruction storage, i.e., laid out and filled by the compiler of the C program. The user is not expected to make any assumptions about its structure, using only addresses computed at compile time. Therefore, we can base it anywhere. Furthermore, mapping memory at address 0 would require root privileges in most modern Linux configurations.

DTCM is the core-local data memory, not accessible from any other core. Note that in the POSIX thread model, all threads within a process share their memory space. Therefore, we must run each core as a separate process to separate their DTCMs while mapping them at the same address. The stack and static data section of the program also go into the DTCM on the actual SpiNNaker board, but for simplicity the reference implementation handles them in the usual way for Linux i386 binaries. That is, the DTCM is only used by `spin1_malloc()`.

The SDRAM is shared by all cores on a chip. Given the above, each of these cores runs in a separate process, so we must set up the shared SDRAM manually. We can easily achieve this by memory mapping (`mmap()`) either a file or a SHM handle (c.f. `shm_open()`) at the absolute address. This is not, actually portable usage, but works very well on Linux. Our reference implementation uses a file, which has the added advantage that the memory contents can be inspected by the user at runtime using a tool of her choice.

The SYSRAM is very much like the SDRAM, so we handle it in the same way. Our implementation does not take any steps to emulate its contents, but this would be easy to add.

The layout sketched in Table 1 requires one kludge not directly related to SpiNNaker: the default layout of ELF binaries on i386 Linux puts the text section (i.e., the program code) at address 0x400000. This is the same address as the DTCM. Therefore, we instruct the linker to use another address. For the GNU toolchain, it suffices to pass

```
-Wl,-Ttext-segment=0x30000000
```

to all `gcc` invocations concerned with linking. The address is essentially arbitrary as long as it does not collide with any SpiNNaker usage.

2.2 Communications

Like almost everything else, we handle the communication at the API level instead of the true system implementations. This means we need to handle two packet types: multicast (32 or 64 bits) and SDP (theoretically any length, but currently limited to headers plus 256 bytes). That is, we completely ignore the other routing modes implemented by the hardware, such as point-to-point, and handle SDP packets in one piece.

We can then use any datagram-based transport layer for the actual message passing. Our reference implementation sends both the program's and its internal communication over the same UNIX domain datagram sockets. UDP would be another obvious choice. For simplicity, all routing happens in a single control process. Each emulator process (one per core) communicates with the control process over one socket.

For this communication we use an ad-hoc packet format with the following message types:

MC Multicast. The sending core's process passes the multicast packet to the control process, which sends it to the appropriate core's input queues.

SDP SpiNNaker Datagram Protocol packet. Again, the sender just hands it off to the control process, which sends it to the receiving core. The control process also listens to a UDP socket for SDP-over-UDP traffic, and likewise sends Internet-addressed packets over UDP.

Route Used to pass `spin1_set_mc_table_entry()` calls to the control process. The payload consists of the arguments of the call; the control process keeps track of the routing table.

Exec Signals to a core process that it should start a client program. See 2.4 for details on how program startup is implemented.

Startdata Informs the core process of its parameters immediately after `execve()`. Currently it transmits chip and core number, and the location of the files used for memory mappings (see Section 2.1).

Ping and Pong are used to implement the optional fine-grained scheduling, see Section 2.5.

The reference implementation sends everything over the same sockets, but makes sure to use a blocking `send()` where control packets are concerned, so as to avoid losing them. This will naturally also flush any queued MC/SDP packets. For greater simulation accuracy, the emulator could use two sockets per core process.

Both core (in a separate thread from the user program, so as to appear "interruptible") and control processes use a `select()` loop to wait for I/O on the sockets from the control/cores, respectively.

2.3 Emulated API

As discussed in Section 1.2, the SpiNNaker API consists mainly of an event loop. The user installs handlers (callback functions) for the events, and calls

`spin1_start()`. The ARK then sleeps until an event happens, at which time it calls the event handler.

The same can be achieved under UNIX using `select()` to wait for new input to arrive. There are two catches.

First, the API provides for a distinction between *queueable* and *immediate* events. The latter essentially run during the interrupt handlers that triggered the event, allowing the user program to, e.g., receive multicast packets while a long-running computation is going on. We emulate this by running the `select()` loop in a separate thread. Even if the program is executing some other event handler, the `select()` thread can concurrently receive a packet and execute the user's handler.

This leads to the second problem: if the program thread is sleeping when the `select()` thread receives new events, we need to wake it up. We achieve this by sending it a signal, at which point it will re-examine its event queues.

2.4 Program startup and `ytag`

In the first phase of a SpiNNaker program, the user customarily uses a program called `ytag` to interface with the SC&MP. This can be used to load data to a memory address, load and start user programs, and even (at runtime) inspect memory. In our reference implementation, the control process understands a subset of the `ytag` language that can load data and start programs. This lets the user start their SpiNNaker program in the same way as on the real board.

To load data, the control process maps the memory of the chosen core and then reads data directly into it.

Starting programs is slightly trickier. In theory we would only need to `fork()` each core from the control process, so that the user program can be started directly. However, this has some downsides: the user could then not run different programs on the various cores. In addition, POSIX disallows threaded programs from doing pretty much anything useful between `fork()` and `execve()`.

To avoid these problems, the control process is a separate – unthreaded – program. Upon startup, it sets up all sockets and runs `fork()` for each core; the core processes then wait for the "exec" packet. When the control process receives a `ytag` command that runs a user program, such as `ff`, it sends an "exec" packet containing the program to execute, followed by a "startdata" packet with the necessary program state.

This strategy hinges on passing the communications socket to the child in some way. We achieve this by using `dup2()` to ensure the socked is a fixed file descriptor, and simply inheriting it across the `execve()` call.

2.5 Scheduling

We also experimented with programs that are not solely event-driven, but also execute a fixed workload over and over as long as no other events arrive, broadcasting the results of their computation to the other cores. On the SpiNNaker hardware this gives good results, since all cores execute at the same time.

However, the emulation ran into scheduling issues: Linux tends to schedule threads for a comparatively long time (usually 10ms) before switching to the next one. This means that on the currently very widespread 2- and 4-core

systems, only a handful of threads execute for long enough to fill their output buffers and lose packets.

To alleviate this issue, we implemented an optional fine-grained scheduling feature. It aims to ensure that the execution of each core progresses roughly synchronously. Since this is not possible (unless you have a 64-core system at your disposal), the control process instead signals a random core process with a "ping" packet. The core process then executes a single queueable event handler, and sends back a "pong" packet. The control process reads the "pong", and again signals a random core process with a "ping", etc.

This gave very good results even for programs relying heavily on the highly parallelized nature of the SpiNNaker hardware. Naturally it also results in a huge loss of throughput.

3 Discussion

Using our reference implementation, we have successfully executed several SpiNNaker programs – *unmodified* – on an ordinary laptop with a dual-core processor running Linux. We simply compiled them for execution on a 32-bit x86 CPU, and linked them to the emulator. This emulation scheme has many advantages for debugging, some of which we discuss here.

- The program can be emulated without any modification, which greatly simplifies the workflow. The user can interchangeably run on the SpiNNaker hardware and debug on a computer.
- The emulation requires no special toolkit, such as cross-compilers. The system's usual C compiler can be used (with a flag for 32-bit compilation on 64-bit x86 systems).
- All debugging tools and conveniences available to a Linux C programmer can be applied directly. For example, it is trivial to attach GDB to a core to debug its execution, or to the control process to trace message routing. Similarly the entire emulation can be run under runtime inspection tools like Valgrind to find many common problems.
- The core event loops and control process can be instrumented for more debugging aids, such as: tracing the message routing without a debugger, seeing which events are dispatched, etc.

As of this writing, suitable computers are also far easier to obtain than actual SpiNNaker hardware, making the emulator a good choice for those not lucky enough to have a SpiNNaker board available to them all the time.

A very useful next step would be the implementation of a way to get at the packets in a proper packet sniffer (e.g., Wireshark). This could be implemented either by switching all communications to UDP on the loopback interface, or by writing out a proper packet dump in libpcap format.

4 Conclusion

The SpiNNaker platform is a promising approach to increase the parallelism in neural network simulations. Thanks to the event-driven SpiNNaker API/ARK

[SPGF11], it is programmable in standard C in a setting familiar to all Unix programmers. However, the resulting programs are incredibly hard to debug because of the limited ways (and bandwidth) in which a running program can be interacted with. We have shown a framework that emulates the API/ARK on Linux, allowing the user to run a SpiNNaker program unmodified on an ordinary computer. This lets them work in a familiar environment with a broad range of tools – debuggers, memory inspectors, etc. – commonly used by C programmers. It also opens up SpiNNaker development to users who do not have the SpiNNaker hardware available for debugging use.

Acknowledgments

We would like to thank the SpiNNaker group in general, and Thomas Sharp and Luis A. Plana in particular, for constructive discussion and advice.

References

[PFT+07] L.A. Plana, S.B. Furber, S. Temple, M. Khan, Y. Shi, J. Wu, and S. Yang. A gals infrastructure for a massively parallel multiprocessor. *Design & Test of Computers, IEEE*, 24(5):454–463, 2007.

[RGD+11] A. Rast, F. Galluppi, S. Davies, L. Plana, C. Patterson, T. Sharp, D. Lester, and S. Furber. Concurrent heterogeneous neural model simulation on real-time neuromimetic hardware. *Neural Networks*, 2011.

[SPGF11] T. Sharp, L. Plana, F. Galluppi, and S. Furber. Event-driven simulation of arbitrary spiking neural networks on spinnaker. In *Neural Information Processing*, pages 424–430. Springer, 2011.