Streamlining SpiNNaker

David R Lester, Michael Hopkins, Andrew Rowley, Alan Stokes,
Oliver Rhodes, Andrew Gait, Donal Fellows, Christian Brenninkmeijer,
James Garside, Luis Plana, Simon Davison, Steven Furber

July 30, 2018

Chapter 1

Racy Randoms: Pseudo Random Numbers and Discrete Distributions for Monte-Carlo Simulations

Streamlining SpiNNaker, Part I

There are now very good Pseudo-Random Number Generators (PRNGs) for Monte-Carlo Simulations, some of which we will describe in this paper. Nevertheless, *efficiently using* these uniformly distributed PRNGs is equally important. Originally this work arose from a desire to improve the performance of our Neuromorphic Simulation Engine: SpiNNaker, and in particular pseudo-random numbers drawn from a Poisson Process with rate parameter λ, used to simulate 'shot noise'. We will start with this problem, show the techniques used to arrive at a better solution to the problem, and finally discuss how the technique we present can be generalised to generate pseudo-random numbers for other discrete distributions.

1.1 Shot Noise

Shot Noise occurs in nature due to the quantisation of energy in an electrical or other system. The statistical assumption made is that the events being measured, occur independently with a fixed rate. Thus the number of emitted particles from a large quantity of a long half-life radioactive isotope could be modelled in this way.

In neuromophics we would be modelling the electrical noise in a brain-like network as shot noise, representing the background spiking events of other neurons. The number of spikes delivered in a particular time interval is then a rate parameter: λ. If these events are independent then they will can be modelled as a Poisson Process with rate parameter λ. In many of our examples, we'll take $\lambda = 1.6$. Now the Poisson distribution can be defined by the *probability mass function* or *pmf*, where the probability of k events occurring is given by the equation:

$$P(k) = \frac{e^{-\lambda}\lambda^k}{k!}$$

Michael Hopkins has produced a simple implementation of this based on that of Knuth.

```
uint32_t poisson_dist_variate_exp_minus_lambda(
        uniform_rng uni_rng,
        uint32_t* seed_arg,
        ulf_t exp_minus_lambda)
{
    ulf_t p = 1.0ulr;
    uint32_t k = 0;

    do {
        k++;
        p = p * ulrbits(uni_rng(seed_arg));
    } while (p > exp_minus_lambda);
    return k - 1;
}
```

The calculation of $e^{-\lambda}$ takes place seperately as it takes about 50 cycles.

1.2 Instrumentation

The first place to begin is by measuring the existing system; *i.e.* both the PRNG and the Poisson distribution part.

Compiled with `-Ofast` we find that the JKISS-64 of Marsaglia takes just 21 clock cycles on SpiNNaker-1. The C code for this PRNG is:

```
uint32_t mars_kiss64_simp(void)
{
    static uint32_t
        x = 123456789,
        y = 987654321,
        z = 43219876,
        c = 6543217; /* Seed variables */
    uint64_t t;
    x = 314527869 * x + 1234567;
    y ^= y << 5;
    y ^= y >> 7;
    y ^= y << 22;
    t = 4294584393ULL * z + c;
    c = t >> 32;
    z = t;

    return (uint32_t) x + y + z;
}
```

A decompilation with `arm-none-eabi-objdump -Dax` gives the following ARM code:

```
0000036c <mars_kiss64_simp>:
     36c:       ldr     r2, [pc, #80]   ; 3c4 <mars_kiss64_simp+0x58>
     370:       push    {r4, r5, lr}
     374:       ldr     r3, [pc, #76]   ; 3c8 <mars_kiss64_simp+0x5c>
     378:       ldr     r4, [r2, #12]
     37c:       ldr     lr, [r2, #8]
     380:       ldr     ip, [r2]
     384:       ldr     r0, [pc, #64]   ; 3cc <mars_kiss64_simp+0x60>
     388:       mov     r5, #0
     38c:       ldr     r1, [pc, #60]   ; 3d0 <mars_kiss64_simp+0x64>
     390:       umlal   r4, r5, r3, lr
     394:       ldr     r3, [r2, #4]
     398:       mla     r1, r0, ip, r1
     39c:       eor     r3, r3, r3, lsl #5
     3a0:       eor     r3, r3, r3, lsr #7
     3a4:       eor     r3, r3, r3, lsl #22
     3a8:       add     r0, r1, r4
     3ac:       str     r5, [r2, #12]
     3b0:       str     r4, [r2, #8]
     3b4:       str     r1, [r2]
     3b8:       add     r0, r0, r3
     3bc:       str     r3, [r2, #4]
     3c0:       pop     {r4, r5, pc}
     3c4:       .word   0x00400000
     3c8:       .word   0xfffa2849
     3cc:       .word   0x12bf507d
     3d0:       .word   0x0012d687
```

How can this possibly execute in just 21 cycles when there are 22 instructions, quite a few of which take two or more cycles to complete? The key is to ensure that all the calls to the function are inlined, and are localised. Then the compiler will exploit the fact that the constants and state are not repeatedly re-loaded to registers, but instead re-use them, as is. If we ignore this advice, then Michael Hopkins has previously reported that this code takes 44 cycles.

In conjunction with the implementation of Poisson given above this takes – on average – 211.09 cycles to execute. The minimum time was 89 cycles, and the maximum was 689 cycles.

1.3 A New Implementation

Rather than just keep the expression $e^{-\lambda}$ pre-calculated, we can instead pre-calculate a table of cumulative probabilities as follows:

```
//! \brief This array holds a value n at position i, such that
//! whenever a uniformly distributed 32-bit unsigned input x
//! satisfies:
//!
//!        table[i] <= x < table[i-1]
//!                   {with table[-1] notionally being MAXUINT}
//!
//! With the given table entries, then the varible i will be a
//! Poisson PRNG, with lambda = 1.6.

uint32_t fast_poisson_table[]
    = { 3427828354,  // p = 0; 3427828354.0287867
        2040406047,  // p = 1; 2040406046.8874736
        930468201,   // p = 2; 930468201.1744232
        338501350,   // p = 3; 338501350.1274629
        101714610,   // p = 4; 101714609.7086788
        25942853,    // p = 5; 25942852.77466787
        5737051,     // p = 6; 5737050.925598297
        1118582,     // p = 7; 1118581.931525252
        194888,      // p = 8; 194888.1327106422
        30676,       // p = 9; 30675.90181026706
        4402,        // p = 10; 4401.944866207042
        580,         // p = 11; 580.2784016164973
        71,          // p = 12; 70.72287300442295
        8,           // p = 13; 8.008346406013947
        1,           // p = 14; 0.8409719376243779
        0            // p = 15, 0.0
      };
```

This table can be easily constructed at compile-time if λ is also known at compile-time. It is even more useful if a number of different poisson noise sources are all using the same parameter λ: then they can share the table. A sketch of a suitable python script is:

```
import numpy as np
```

```
from scipy.stats import poisson

# To generate a table for 8-bit PRNG
lambda_ = 1.6

i = 0
n = (1 - poisson.cdf (0, lambda_)) * (256)
while n > 0.1:
  print n
  i = i+1
  n = (1 - poisson.cdf (i, lambda_)) * (256)

# To generate a table for 32-bit PRNG
i = 0
n = (1 - poisson.cdf (0, lambda_)) * (1024*1024*1024*4)
while n > 0.1:
  print n
  i = i+1
  n = (1 - poisson.cdf (i, lambda_)) * (1024*1024*1024*4)
```

Given a uniformly distributed number `n`, we have merely to search the table for an index `i` such that:

$$\texttt{table}[\texttt{i}] \leq \texttt{n} \wedge \texttt{n} < \texttt{table}[\texttt{i} - 1]$$

This can be accomplished by the following simple linear search:

```
static inline uint32_t fast_poisson (uint32_t* base, uint32_t key)
{
    uint32_t offset = - ((((uint32_t)base) >> 2) + 1);

    while (*base++ > key);

    return (((uint32_t)base >> 2) + offset);
}
```

This generates the following ARM assembler:

```
00000488 <fast_poisson>:
     488:       e1a03120        lsr     r3, r0, #2
     48c:       e3e02001        mvn     r2, #1
     490:       e0632002        rsb     r2, r3, r2
     494:       e4903004        ldr     r3, [r0], #4
     498:       e1530001        cmp     r3, r1
     49c:       8afffffc        bhi     494 <fast_poisson+0xc>
     4a0:       e0820120        add     r0, r2, r0, lsr #2
     4a4:       e12fff1e        bx      lr
```

Although this is a linear search the length of each loop is just three cycles – less the conditional branch instruction which is itself also three cycles, when the branch is taken. Unrolling the loop 16 times, gives a total code length of $14+16\times 3 = 62$ instructions. It may appear poor coding to have a linear search, where a binary search would take only four iterations, but those iterations are more complicated and the execution time rises. Furthermore with $\lambda = 1.6$ over half the results are either 0 or 1, and thus occur early in the search. There is in fact a switch-over point where a combination binary-linear seach will be fastest.

When combined with the JKISS-64 PRNG we now have a Poisson PRNG which takes 37.87 cycles on average, with an observed minimum of 33 cycles, and an observed maximum of 57.

So far, so good.

1.4 Not So Rare Events

> Rather than simply blaming the victims for carelessness, the horses for viciousness, or the generals for incompetent leadership, the mathematician Ladislaus Bortkiewicz, a German of Polish descent, studied the death rates over a 20 year period (18751894). In a groundbreaking book entitled the Law of Small Numbers (a phrase that still resonates in statistical circles) 3, he showed that the data followed the Poisson distribution

`https://onlinelibrary.wiley.com/doi/full/10.1111/anae.13261`

It is worth asking how rare the 'rare events' we are considering actually are. In the complete SpiNNaker system we have 1,000,000 cores each simulating up to 256 neurons, and each neuron will be sampling the random number stream 10,000 times every second (for real-time performance). If the random number stream is a 32-bit stream, and we use the above technique to find a suitable Poisson number, then we will get 640 occurences of 0 from the random number stream, all of which in the above example translate into the Poisson output of 15. Obviously we'd expect at least some of these to be 16, 17 and so on, with decreasing probability.

The solution is to generate a further look-up if a 0 is returned by the uniform PRNG. The key advantage of this is that we only need the extra 32-bits in the extremely rare case of 0 which occurs with probability 2^{-32}. In all other cases the first 32-bits suffices. Note that it is possible to get greater distributional accuracy by using a further lookup in the edge-cases of the first look-up, *i.e.* when the table entry precisely matches the key being searched.

This suggests an alternative way to calculate Poisson PRNGs . . .

1.5 Using smaller chunks of the 32-bit Uniform PRNG

What happens if we use the 32-bit number to generate *four* Poisson values, using a look-up table based on 8-bit values? Is it quicker? Yes it is. Using this technique, it is possible to drive the average time taken down to 27 cycles, though the worst-case is now over 100 cycles.

However, another possibility now arises since the total number of values being considered using a random byte is only 256: use a look-up table to record the poisson value for 255 of them, and drop into our original code if the random byte is precisely 0[1]. The code needed is now:

```
//! \brief Generate a poisson random value for the random byte "key"
//! using the random number stream "prng", returning the result in
//! "result", and using two tables. The returned value is the new
//! random number stream, less any values consumed by this routine.

uint32_t* __fast_poisson_byte (uint32_t* prng,
      uint32_t key, uint32_t* result,
      uint32_t* table_8, uint32_t* table_32)
{
    if (key == 0) {
        *result = fast_poisson (table_32, *prng++ >> 8);
return (prng);
    }

    *result = table_8[key];

    return (prng);
}
```

Generating a full four values is then:

```
uint32_t* fast_poisson_bytes (uint32_t* prng, uint32_t keys, uint32_t* result,
      uint32_t* table_8, uint32_t* table_32)
{
    prng = __fast_poisson_byte (prng,  keys       & 0xff,
                                result++, table_8, table_32);
    prng = __fast_poisson_byte (prng, (keys >>  8) & 0xff,
                                result++, table_8, table_32);
    prng = __fast_poisson_byte (prng, (keys >> 16) & 0xff,
                                result++, table_8, table_32);
    prng = __fast_poisson_byte (prng, (keys >> 24),
```

[1]Note that we have a slightly less accurate Poisson Distribution because of the granularity of using 255 entries; this too, could be corrected by special-casing – though at the cost of increased execution time.

```
                                        result,   table_8, table_32);

    return (prng);
}
```

The `table_8` looks as follows:

```
uint32_t __fast_poisson_byte_table[]
   = {6, 5, 5, 5, 5, 5, 4, 4,
      4, 4, 4, 4, 4, 4, 4, 4,
      4, 4, 4, 4, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 3,
      3, 3, 3, 3, 3, 3, 3, 2,
      2, 2, 2, 2, 2, 2, 2, 2,

      ...
      0, 0, 0, 0, 0, 0, 0, 0};
```

Now the execution time – including the time taken to generate JKISS-64 random numbers used – is 13.89 per Poisson Value generated and stored. The best case for generating four values is 55; the worst case is 92.

This result – of just under 14 cycles – is comparable to the execution time of our neuron model, and is thus acceptable for now.

To be Continued .... Maybe!!

I suspect that careful assembler code could squeeze a few extra cycles out of the table lookup, especially if we make the small table be byte-sized instead of word sized. (Reason: GCC is not so good with array-indexing for non-word data.)

1.6 Other Distributions

It should be obvious, but we high-light here that this table-driven approach can be used to generate random numbers for other discrete distributions, simply by changing the tables used, *e.g.* the Binomial distribution for small values of N, hypergeometric, *etc, etc.*

1.7 Conclusions

- Because the ARM is a register-register machine, it is extremely important to ensure that once registers are loaded, we make the maximum use of them.

  In particular, it makes sense to generate a lot of JKISS-64 random numbers in one go; then use those numbers to generate Poisson Events (again in one go); and then model a lot of neurons in one go.

This is more efficent that modelling a neuron, generating the shot noise, which in turn calls the random number generator.

- Fast execution of SpiNNaker involves pre-generating at compile-time, tables of results in preference to execution-time calculation.

- The obvious algorithm selection for hardware is to search the user-supplied 16-entry `fast_poisson_table` (with 32-bit values) in parallel. If the table is insufficiently large – as it will be with $\lambda = 3.0$ – then we drop into software to do the rare tail cases, corresponding to more than 15 events per time interval.

Chapter 2

Nippy Neurons: Fast Integrate-and-Fire Neurons

Streamlining SpiNNaker, Part II

The provision of different Neuron Models – and especially user-defined or modified models – is a critical factor for the acceptance of a neural simulator. Nevertheless, in a neuromorphic setting, where there is less scope for highly-detailed models, we need to ensure that we do the simple things well. In particular, it is one of the key goals to ensure that the very simple leaky-integrate-and-fire (LIF) neural models execute as fast as possible.

In addition, a detailed consideration of the mechanisms needed to improve the performance of LIF neurons will feed into better code for other, more complicated, neuron models.

2.1 LIF ODE

A simple ODE system for the neuron membrane voltage (v) and exponentially-shaped current post-synaptic potential (p) is:

$$\begin{array}{rcrcrcr} v' & = & a_{00}v & + & a_{10}p & + & b_0 \\ p' & = & & & a_{11}p & & \end{array}$$

As the matrix-like subscripts suggest, this ODE can also be expressed in matrix form:

$$\tfrac{d}{dt}\,\mathbf{x} = \begin{pmatrix} v' \\ p' \end{pmatrix} = \begin{pmatrix} a_{00} & a_{10} \\ 0 & a_{11} \end{pmatrix} \cdot \begin{pmatrix} v \\ p \end{pmatrix} + \begin{pmatrix} b_0 \\ 0 \end{pmatrix} = A\mathbf{x} + \mathbf{b}$$

And therefore this is a *linear* ODE system. And *they* have an exact solution, given a starting value for the vector $\mathbf{x} = \mathbf{x}_0$. The solution to the *Initial Value Problem* (IVP):

$$\frac{d}{dt}\,\mathbf{x} = A\mathbf{x} + \mathbf{b}, \text{ with } \mathbf{x}(t_0) = \mathbf{x}_0$$

at time $h + t_0$ is:

$$\mathbf{x}(h + t_0) = e^{hA}\mathbf{x}(t_0) + h\phi_1(hA)\mathbf{b}$$

Fortunately the matrix exponential and the vector $\phi_1(hA)\mathbf{b}$ can now be easily and accurately determined by Nick Higham's Algorithm and Saad's Trick. Higham's Algorithm is available in the python `scipy.linalg` library. Saad's Trick is worth mentioning; if we have a matrix exponential function, then we can construct both e^{hA} and $\phi_1(hA)\mathbf{b}$ simultaneously. The observation is that:

$$\exp\left(\begin{array}{c|c} hA & \mathbf{b} \\ \hline 0 & 0 \end{array}\right) = \left(\begin{array}{c|c} e^{hA} & \phi_1(hA)\mathbf{b} \\ \hline 0 & 1 \end{array}\right)$$

In short, just attach the vector $\mathbf{b}$ to the right hand side of the matrix hA, attach a row of zeros below, and $\phi_1(hA)\mathbf{b}$ will be produced using the standard matrix exponential routine.

It is worth pointing out that this routine can go wrong when the original matrix A is ill-conditioned. A is ill-conditioned if either it or its inverse are "nearly singular", *i.e* have determinant near zero. So, first check for a non-zero determinant, and then check the condition number. This has been observed in a neural setting by Markram and Tsodyks, who advise against giving the neuron and the psp the same time-constant. In our terms this amounts to making $a_{00} = a_{11}$, and thus giving a singular matrix.

A python sketch for this calculation is:

```
import numpy as np
from scipy.linalg import expm

h          =    0.1 # Timestep
v_rest     =  -65.0 # Resting voltage
tau_m      =   10.0 # Membrane time-constant
tau_syn    =    2.0 # PSP time-constant
c_m        =  250.0 # Membrane capacitance

mtx = array ([[-1.0/tau_m,   1.0/c_m,     v_rest],
              [         0,  -1.0/tau_syn,      0],
              [         0,          0,         0]])
```

Then the expression `expm`$(0.1 *$ `mtx`$)$ $-$ `np.identity`(3) gives the result:

```
   array([[-9.95016625e-03,  3.88204092e-04, -6.46760806e+00],
          [ 0.00000000e+00, -4.87705755e-02,  0.00000000e+00],
          [ 0.00000000e+00,  0.00000000e+00,  0.00000000e+00]])
```

For reasons which will become apparent later, notice that all of these constants are quite small, except the term $\phi_1(0.1A)\mathbf{b}$ which is about -6.4676. The reason is that we are calculating the *change* from the current vector $\mathrm{x}(t)$, and provided that the time step is small enough, the change should be similarly limited. Indeed as $h \to 0$, $e^{hA} \to I$. A more rigourous result comes from theory and states that the solution to a linear ODE system will necessarily be continuous. Expanding out the matrix above we get:

```
v (t+h) = v(t) - 0.00995016625*v(t) - 0.000388204092*p(t) - 0.646760806
p (t+h) = p(t)                       - 0.0487705755*p(t)
```

A naïve C program for the ODE evolution defined above, assuming a struct-like type for the neuron, is then:

```
typedef struct {double p; double v;} neuron_t, *neuron_ptr;

...

void process_neuron (neuron_ptr np)
{
    double p, v;

    p = *np->p;
    v = *np->v;

    v -= v * 0.00995016625;
    v += p * 0.000388204092;
    v -=     0.646760806;

    p -= p * 0.0487705755;

    *np->p = p;
    *np->v = v;
}
```

There are a significant number of problems with this code, which we will subsequently circumvent, but the standout initial problem is that it uses `double`s, and SpiNNaker does not have hardware support for this type!

As an alternative the SpiNNaker team encourages users to use the ISO fixed point arithmetic standard for C, which is supported by `gcc`, and also by the extensions the SpiNNaker team have added to the ARM ACLE "ARM Common Language Extensions" by way of C intrinsics for the special DSP instructions. The drawback with using the ISO fixed point package is that it is not yet well-enough integrated into GCC, and thus can be quite expensive to use. One of the reasons for this expense is that the three `accum` multiplications operate by subroutine call (which adds six cycles), and internally multiply the two 32-bit numbers – as integers – to produce a 64-bit result, which is then arithmetically right-shifted by 15-bits. Done correctly, it is possible to butt these instructions together without interlocks; nevertheless, the multiplication will take at least three cycles (or four if the Most Significant Word is used first). In addition, this code cannot take advantage of the multiply-accumulate for two reasons: firstly the compilation is not smart enough to spot the optimization, but secondly two of the three usages involve *multiply-subtract* which is not available in ARM-9. Changing the multipliers back to negative numbers at least solves the second problem, as we shall see, later in Section 2.3.

2.2 Boring Stuff about our Current Implementation

Blah, blah, blah!

2.3 Exploiting the DSP Instructions

The DSP Instructions available in the `v5te` instruction set include a useful set of fixed point signed multiply, and signed multiply-accumulate instructions. We'll look at the `smlawx` case (`x` is set to one of `t` or `b`).

```
        smlawb r0, r1, r2, r3
```

In the above assembler the effect of this instruction is to multiply the 32-bit signed value in register `r1` by the *lower* 16-bits in register `r2` treated as a signed 16-bit fraction, and then arithmetic shift this resulting product right one bit. This is then added to the accumulator in register `r3` and the result is placed in register `r0`. This instruction takes just one cycle, provided that the result in register `r0` is either used as an input accumulator to another DSP multiply-accumulate in the next instruction, or it is not required for at least one instruction. If the result *is* required by the next non-DSP multiply-accumulate instruction, then this instruction takes two cycles.

The odd behaviour with the extra right-shift by one bit has the apparent effect of making a multiplication by the 16-bit ISO signed fraction (`fract`) -1 (represented by having the bit-pattern `0x8000` in `r2`) give the effect of multiplication by $-1/2$. In other words the available multipliers run from $-1/2$ to $+1/2 - 1/32768$.

```
typedef struct {int p; int v;} neuron_t, *neuron_ptr;

...

void process_neuron (neuron_ptr np)
{
    int p, v;
    int c[4] = {0x0000fd74, // representing -0.00995016625
```

```
                0x00000019, // representing  0.000388204092
                0x0000F384, // representing -0.0487705755
                0x000052c9};// representing  0.646760806

    p = *np->p;
    v = *np->v;

    v  = __smlawb (v, c[0], v);
    v  = __smlawb (p, c[1], v);
    v -= c[3];

    p  = __smlawb (p, c[2], p);

    (*np++)->p = p;
    (*np++)->v = v;
}
```

Blah, blah, blah! Performance of this and other approaches.

2.4 Identical Leaky-Integrate-and-Fire Neurons

A number of "tricks" have been rolled into this clever code.

The first important point to note is that setting the neurons' threshold to 0, by treating all voltages relative to v_Θ means that we can combine a standard ARM data-processing instruction with the threshold detection, by – for example – substituting the `subs` instruction for `sub`. This works because the `Z` and `N` flags are set in relation to 0, whereas comparing with $v_\Theta \neq 0$ would require an explicit `cmp` instruction.

With this decision, we have now freed up half the available voltage – that greater than or equal to 0 – for other purposes. The neurons we are looking at in this paper have a refractory period, where the neuron counts out n time-steps before the voltage is reset (to v_{reset}). Once more, having the refractory counter count down towards 0 and detecting the need for a reset of the voltage when 0 is reached saves unnecessary `cmp` instructions.

$$v = \begin{cases} \text{refractory timer with } v \text{ time-steps remaining} & \text{if } v > 0 \\ \text{refractory period over, reset required} & \text{if } v = 0 \\ \text{normal ODE processing} & \text{if } v < 0 \end{cases} \tag{2.1}$$

We'll now show a first attempt at the ARM assembler for the neuron. We assume that the 16-bit constants for the decays are in registers `r9` and `sl` (also known as `r10`). The reset voltage is in register `fp` (also known as `r11`). The PSP is held in `r8` and the original voltage in `r7`. Calculation of the new voltage occurs in temporary register `r3`.

```
    ldr     r8, [r2]                ; load psp current
    ldr     r7, [r2, #4]            ; load voltage
    ldr     r3, [r1], #4            ; load input to temporary register r3
    add     r8, r3, r8              ; combine psp and input
    smlawb  r8, r8, sl, r8          ; decay combined psp and input
    cmp     r7, #0                  ; test original voltage,
                                    ;  which is refractory if r7 >= 0
    bge     <L_refract>

    smlawb  r3, r7, r9, r7          ; decay voltage (due to voltage value)
    smlawt  r3, r8, r9, r3          ; decay voltage (due to psp value)
    subs    r3, r3, #16777216       ; subtract drift term from voltage
                                    ;  and also setting flags
    bge     <L_spike>               ; spike generated if r3 >= 0
L1:
    str     r8, [r2], #4            ; store psp back now, post-incrementing
    str     r3, [r2], #4            ; store back the newly calculated voltage
                                    ;  into the neuron; post-incrementing
    bx      lr                      ; return

L_refract:
    moveq   r3, fp                  ; if precisely zero, then assign
                                    ;   reset voltage from fp
    subgt   r3, r7, #1              ; if original voltage > 0, then decrement
                                    ;     refractory counter
    b       <L1>

L_spike:
    mov     r3, #20                 ; Set refractory counter to 20
    str     r2, [r0], #4            ; Store neuron pointer value in
                                    ;  spike buffer for later post-processing
    b       <L1>
```

In the most frequent case – without any refractory or spiking operations – this code ought to execute in just 13 cycles; but it doesn't. The problem is that the load and multiply-accumulate instructions both have an *interlock* if their inputs or outputs are still in one of the ARM processor's pipeline stages.

For example the `subs` instruction following the `smlawt` instruction introduces a one cycle delay because register `r3` is not immediately available. Similarly the `add` instruction also cannot access register `r3` since the result of the `ldr` instruction is not immediately available.

A good compiler should be able to rearrange the instructions so that something useful occurs between the generation and usage of a register; but – as Mae West might almost have observed – good compilers are hard to find. This goes *a fortiori* for compilers that know about the DSP instructions!

So, we must revert to the good old Mk 1 "Human" Compiler, to re-order these particular instructions.

Before we do so, there is one other useful trick, and that is to try to combine the effects of the special cases for spiking and refractory periods into a single sub-routine. A quick calculation shows that if the neurons spike at a frequency of 10Hz, have a refractory period of 2ms, and the time-step is set at 0.1ms, then these special cases are: spiking (10), refractory period (200) and ordinary ODE-solving ($10,000 - 210 = 9,790$). In short the common case is ODE-solving and it occurs 97.9% of the time. Thus some small amount of re-calculation within the sub-routine – if needed – will often be worthwhile.

2.5 A Twelve-Cycle LIF Neuron Loop-body

The LIF dynamics for a single neuron update are coded here. This includes the shaping of the PSP or synapse current as well. It is coded in assembler for performance, both execution speed and code density. The resultant assembler has three parts: a header, a body with unrolled loops, and a tail. The header is as follows, and stacks the used registers, as per the EABI protocol. It also loads the constants from the `kp` argument.

```
push    {r7, r8, r9, sl, fp, lr} ; stack used registers
ldr     r9, [r3]                 ; load k0
ldr     sl, [r3, #4]             ; load k1
ldr     fp, [r3, #8]             ; load k2
```

The body consists of as many copies of the following twelve instructions as it is deemed sensible to have. 8, 16 or perhaps even 32 seem sensible.

```
ldr     r3, [r1], #4       ; load input to temporary register r3
ldr     r8, [r2], #4       ; load psp current, post-incrementing
ldr     r7, [r2], #4       ; load voltage, post-incrementing
add     r8, r3, r8         ; combine psp and input
cmp     r7, #0             ; test original voltage,
                           ;    which is refractory if r7 >= 0
smlawb  r8, r8, sl, r8     ; decay combined psp and input
smlawb  r3, r7, r9, r7     ; decay voltage (due to voltage value)
smlawt  r3, r8, r9, r3     ; decay voltage (due to psp value)
str     r8, [r2, #-8]      ; store psp back now; allow for r2 change
subslt  r3, r3, #16777216  ; if non-refractory subtract drift term
                           ;  from voltage and also set flags
blge    <L_special>        ; branch-and-link to special handling
                           ;  if either the original voltage >= 0;
                           ;  or the new voltage is >= 0.
str     r3, [r2, #-4]      ; store back the newly calculated
                           ;  voltage into the neuron; nb r2 change
```

Finally we have the tail part, which consists of the return (including unstacking all the clobbered registers), and the special handling sub-routine.

```
    pop     {r7, r8, r9, sl, fp, pc}; unstack and return.

L_special:
    cmp     r7, #0            ; re-test for refractory period
                              ;  i.e. (original voltage >= 0)
    moveq   r3, fp            ; if precisely zero, then assign
                              ;  reset voltage from k2
    subgt   r3, r7, #1        ; if original voltage > 0, then
                              ;  decrement refractory counter
    bxge    lr                ; if original voltage >= 0, then
                              ;  return

; if we get to here the newly calculated voltage must be >= 0,
; and thus we generate a spike

    mov     r3, #20           ; Set refractory counter to 20
    str     r2, [r0], #4      ; Store neuron pointer value in
                              ;  spike buffer for later post-
                              ;  processing
    bx      lr                ; return to main neuron processing.
```

Sure enough, this routine executes in 12 cycles for ODE processing, 20 cycles for refractory processing, and 23 cycles when a spike occurs. Note that in the last case we have not accounted for calculating the 32-bit spike packet from the neuron address, nor for its transfer onto the network fabric.

Given the previous assumptions above about neuron firing frequency, refractory period, and a 0.1ms time-step, the expected execution time (with the loop unrolled 32 times) is 12.9835 cycles per neuron, on average, including the call/return sequence and the loading of the three constant registers.

Further details of measurements and trade-offs needed here!

2.6 Generalising the LIF Model

Of course not all PyNN scripts have identical LIF neuron parameters. If each neuron has a different parameter set – perhaps even with different ODE parameters – then we will need to provide an extra pointer in the `struct` representing each neuron, to access these different parameters. This will behave in the same way as register `r3` in the previous section, but we will need to load in the constants within the loop for each register.

By calculation (*not measurement*), I estimate this adds an additional 4 cycles to the average case. Note that access to the refractory counter reset value and the reset voltage is only required when a spike occurs, and thus we only need to load the constants corresponding to the matrix multipliers, and the $h\phi_1(hA)\mathbf{b}$, or drift, term.

2.7 Other Neuron Models

As the complexity of the mathematics needed to solve the ODE increases, the squeezing out of a few cycles here and there has a lesser proportional effect. Nevertheless, it is worth looking at the Izhikevich Neuron, which is possibly the next most simple to the LIF model.

More stuff here..

Chapter 3

Snappy Synapses (1) Fast Fixed Synapse Processing *Streamlining SpiNNaker, Part III*

People are often surprised that much of the effort in neural simulations is expended modelling synapses rather than neurons. This should not be *so* surprising, as each neuron is connected – on average – to 10,000 other neurons. Recent research has shown that even in models with synaptic plasticity, *i.e* where the connection efficacy changes as the system learns, there are still five or six times as many fixed synapses as there are plastic synapses (cite Maass).

It therefore makes sense to make this operation as efficient as possible.

3.1 Ring Buffers

The approach taken by SpiNNaker – following Morrison's suggestion which she'd used in NEST – is to provide a circular buffer for the inputs to a neuron. The assumption being made is that synaptic events act in a purely additive, or linear, manner; and that we can therefore treat the action of two similtaneous synaptic events as the addition of their effects.

Conceptually, we use this circular buffer to delay the actions associated for the events by a suitable delay. When the effects reach the front of the queue, we can then pass them on to the neuron ODE solver to manipulate the inputs according to the defining equations. Usually, this amounts to the decay of the value.

More stuff here.

3.2 A Simple Non-Saturating Unsigned Synapse/Ring-Buffer Implementation

We will assume that the synaptic word is laid out with three bit fields. These are a *weight*, a *delay*, and an *address*, and we will lay these fields out in that order from the most-significant to the least-significant end of the word.

A simple C program would be:

```
typedef int* synapse_ptr;
int ring[RING_SIZE];
int time;

...

void process_synapses (synapse_ptr sp, int n)
{
    int w, addr, i;

    for (i = 0; i < n; i++) {
        w = (*sp++);
              // loads synaptic word w

        addr = (w + (time << 8)) & 0xffff;
              // adjusts the address in the bottom 16 bits
              // for the correct delay

        ring[addr] += (w >> 16);
              // adds weight in the top 16 bits of w to
              // ring buffer
    }
}
```

3.3 Boring, Boring, Boring!!

Describe problems above. Measure speed!

3.4 A Five Cycle Synaptic Row (or Rowlet) Processing Loop

This approach is permitted whenever we can afforrd to have 32-bit ring-buffer elements. In this case, we can be sure that adding weights to the ring-buffer will be extremely unlikely to overflow the integer.

With signed weights in the synaptic word (assumed to be 16-bit, but could be larger) and signed 32-bit ring-buffers the basic FIVE*five* instruction sequence is:

```
ldr   lr, [r0], #4           ; Load a synaptic word
add   r10, r2, lr, lsl #23   ; add timer value (top bit masked off)
ldr   r8, [r1, r10, lsr #22] ; use load to add base to shifted index
add   r8, r8, lr, asr #16    ; _arithmetic_ shifted weight _added_.
str   r8, [r1, r10, lsr #22] ; store back result.
```

The synaptic word pointer is register `r0`, the base address of the ring buffer is `r1` and the shifted timer (with its top bits set to the current time) is in `r2`. This shifted timer value is added to the shifted synaptic word, so that any overflow is ignored. Then note use of array indexing to shift the register `r10` back down to the correct position as part of the `ldr` and `str` instructions. This is how we've eliminated masking.

However, there is an obvious problem: the use of `r8` in the `add` instruction, and it's subsequent use in the `str` all cause *interlock* problems. *I think there are other problems too!* Simple re-ordering will no longer suffice in this case, since the order of the instructions is pretty pre-ordained by the program logic! Instead, we use another common trick: interleaving *two* copies of the code, operating with different registers, to mask the interlocks.

```
ldr   r4, [r0], #4           ; Load a synaptic word_0
ldr   r5, [r0], #4           ; Load a synaptic word_1
add   r6, r2, r4, lsl #23    ; add timer value (top bit masked off)
add   r7, r2, r5, lsl #23    ; add timer value (top bit masked off)
ldr   r8, [r1, r6, lsr #22]  ; use load to add base to shifted index
ldr   r9, [r1, r7, lsr #22]  ; use load to add base to shifted index
add   r8, r8, r4, asr #16    ; _arithmetic_ shifted weight _added_.
add   r9, r9, r5, asr #16    ; _arithmetic_ shifted weight _added_.
str   r8, [r1, r6, lsr #22]  ; store back result.
str   r9, [r1, r7, lsr #22]  ; store back result.
```

Now all instructions operate with no interlocking. Admittedly we have to do two synaptic events as a pair, but hopefully that's not too onerous a constraint to achieve.

3.5 A Six-and-a-Half Cycle Synaptic Row (or Rowlet) Processing Loop

In this version we will restrict the ring buffer elements to unsigned half-words. In this case we *have* to take the possibility of saturation seriously. In this case the basic loop body is:

```
ldr     lr, [r0], #4           ; Load a synaptic word
add     r10, r2, lr,  lsl #23  ; Add the timer (with the extra bit
                                      ;   dropping off the top)
    add     r10, r1, r10, lsl #22     ; Add the ring-buffer base register,
                                      ;   and shift right
    ldrh    r8, [r10]                 ; load the ring buffer element
    subs    r8, r8, lr, lsr #16       ; _subtract_ weight from ring element
    blmi    <saturation>              ; if the result is negative: saturate
    strh    r8, [r10]                 ; store the ring buffer element
```

In the case of half-word load and store operations we no longer have the convenience of a shifted array-indexing operation: we have to do this manually with another add.

The other difference is that we have replaced the `add` operation of the previous example, with a `subs`; why is this? Well, we've inverted the meaning of the weights. Now a weight of zero 0 is represented by the hex value `0xffff`, and the maximum weight of 65535 is reprresented by `0x0000`. Why?

The answer is that it is exteremely convenient to detect saturation by the result of the arithmetic operation turning negative. As we shall see, this permits some level of reconstruction, should saturation arise.

The code above is also subject to memory interlock, but now the problem is even worse: each load and store instruction, may require *two* interlock cycles, because there is extra pipeline delay caused by the required shifting of the half-word result.

We try our previous trick, and combine *two* saturation detections into one sub-routine call — in the hopes that saturation is the rare case.

```
    ldr     lr, [r0], #4        ; load w0
    ldr     ip, [r0], #4        ; load w1
    add     r10, r2, lr, lsl #23 ; calculate address: a0
    add     r10, r1, r10, lsr #22
    add     r11, r2, ip, lsl #23 ; calculate address: a1
    ldrh    r8, [r10]           ; load ring element 1
    add     r11, r1, r11, lsr #22
    ldrh    r9, [r11]           ; load ring element 2
    subs    r8, r8, lr, lsr #16  ; add to ring element 1
    strh    r8, [r10]           ; store ring element 1
    subpls  r9, r9, ip, lsr #16  ; add to ring element 2
    strh    r9, [r11]           ; store ring element 2
    blmi    <saturate>          ; fix-up if either sum saturated
```

Notice that if the first subtraction causes a saturation we *do not* perform the second subtraction; it is left to the saturation routine to fix-up. Notice also, that if we had used the carry flag to spot saturation it is nowhere near as easy to fix-up the result in the event that some of the registers are re-used.

The code for the saturation routine is:

```
    pop    {r3}               ; pop the saturation counter
    cmp    r8, #0             ; is r8 < 0?
    movmi  r8, #0             ; if so: set the value to the maximum (=0)
    addmi  r3, r3, #1         ; increment the saturation counter
    strmih r8, [r10]          ; store back the result, over-writing -ve
    submi  r9, r9, ip, lsr #16 ; the subtraction will not have
                              ;  happened if the first subtraction
                              ;  was negative, so re-do it ...
    cmp    r9, #0             ; is r9 < 0?
    movmi  r9, #0             ; if so: set the value to the maximum (=0)
    addmi  r3, r3, #1         ; increment the saturation counter
    strmih r9, [r11]          ; store back the result, over-writing -ve
    push   {r3}               ; save the saturation counter.
    bx     lr                 ; return.
```

I strongly suspect this may be the very first time anyone has ever used the instruction: `strmih`!

As we can see, this interleaved verrsion of our synapse processing code ought to run in 13 cycles – which it does – meaning that each non-saturating synaptic event will run in just 6.5 cycles, once more, providede we do the processing of these in pairs.

I believe that there is a 25 cycle code sequence which will handle *four* synaptic events at a time – once more commoning up the effects of saturation into just one call to a fix-up routine. The major issue is the need to preserve the intermediate registers *and* provide enough regsiters to control the computation (counters and such like). If this should prove practical, then we can drive the execution cost of the common case where we are using unsigned 16-bit ring buffers down to an amortized 6.25 cycles per synaptic event. For comparison the current code takes (whatever it is), but the code I gave Jamie took 8 or 9 cycles, and I *think* changes to `gcc` have added a bit of subsequent padding here.

Chapter 4

Snappy Synapses (2): Fast Synaptic Rows and Rowlets

Streamlining SpiNNaker, Part IV

This is the much discussed division of an entire synaptic row into chunks whose length corresponds to the available ring-buffer space.

We will also discuss fast methods for indexing into the SDRAM space where the rows are stored.

4.1 Decoupling Spike Handling

There are now *two* processes in play: getting the address of the relevant first synaptic rowlet, and then organising that the rowlets are pre-fetched (and maybe, even copied into the ring buffer) prior to the time-step in which they are actually needed.

This then results in significantly less time pressdure on the DMA engine. The assumption I'm making is that we postpone processing the pre-fetched rowlets until we have reset the ring buffer elements used as inputs in the current time-step – thereby freeing what will become the *last* delay slot of the ring buffers in the next time-step.

Chapter 5

Reducing the Run Time Executive: Speeding up Context-Switching, Scheduling, and Interrupt Handling

Streamlining SpiNNaker, Part V

A major problem with ARM processors is that it takes time to load and store the data needed between registers and memory whenever there is a context switch. This context switch need not necessarily occur when there is an interrupt, it also occurs when there are function calls requiring an extensive change to the "working set" stored in the registers.

The contention put forward here is that it is advantageous if the scheduler controls which actions are performed when, since at each decision point it can chose to make use of pre-loaded registers, thereby continuing with its current actions, or alternatively chose to undertake a higher priority task requiring a change of registers.

Keeping the DMA engine well-stoked is key to high performance, since it can be thought of as a "dumb" parallel processor, operating independently of the CPU, and capable of only block copying memory.

Event-based programming requires us to undertake an action in response to each "event". In SpiNNaker terms an event is usually thought of as receipt of an interrupt.

When an interrupt occurs, enough registers to service the interrupt are pushed onto the stack, new registers are set up (if needed), and at completion of the task associated with the event, this activity occurs in reverse, where registers that need to be preserved between events are saved to memory, and the original registers restored for the original – interrupted – task to continue.

As we have seen, many tasks can be performed in under 32 cycles, which is the worst-case push/pop time for all 16 registers to be saved and restored. If you are able to accept Lester's Aphorism:

> 'Every basic task you'll ever want to do on SpiNNaker can be accomplished – or at least approximated – in fewer than 32 well-chosen ARM Instructions . . .
>
> . . . provided the registers are pre-loaded!'

then this suggests that responding to events should instead merely *note* that the event has occurred, and save up the processing of events until such time as it makes time-economic sense to switch context.

Indeed, if we accept that doing a two-hundred-plus-cycle chunk of computation in one go without interruptions makes sense – equivalent to a granularity of one per cent of the available time budget at 0.1ms – then we might make more use of polling and less use of interrupts proper. (Reference transputer, with its straight line uninterruptable execution, and with only jumps caused context switches).

5.1 Priorities

The following tasks need to be undertaken in most-important-first order.

1. Move incoming spikes into a buffer to free up the SpiNNaker network fabric.

2. Ensure that the DMA engine has been given all available DMA requests.

3. Process Synaptic Rows, so as to free up DTCM for more DMAs.

4. (Optionally, if there are plastic synapses) Ensure Write-backs are prioritised if there is insufficient space to receive more DMAs.

5. Process Neurons.

6. (Optionally) Pre-order the DMA of the rowlets required in the *next* time slot.

7. If all else fails, calculate some more random numbers!

8. If even the PRNG storage is full: go into *Wait For Interrupt* mode.

5.2 Incoming Spike Handling

Responding quickly to incoming spikes is vital. We therefore continue to use the FIQ interrupt for this purpose. Hand-crafted assembler may improve the performance by avoiding the eight registers shared with the non-FIQ code. (Inspection of code required).

5.3 The Care and Feeding of the DMA Engine

Having the control of the DMA in just one place ensures that we can more easily keep tabs on requests and their completion. One of the other tasks listed in Section 5.1, can intermittently poll to see if there has been a DMA completion. If not, it can continue; if there has, it can decide whether it is worth flushing the registers to process the transfered data, or instead whether it has enough spare registers to organise the refilling of the DMA pipeline itself, before continuing.

5.4 Pre-emptive DMAs of rowlets for the next phase

One task mentioned under the priorities heading is to pre-order the rowlets required for the next time-step. Under certain circumstances we can even start processing these!

The requirements are

- That all neurons have been processed. (This ensures that the row of the ring buffer used as inputs in the current time-step are no longer subject to update. We must ensure that they've been zero-ed, by the way!)
- That there is sufficient time left in the current time-step to make this activity worthwhile.
- That no newly received spike will interract with the ring-buffer. We'll come back to this, as this condition can be relaxed slightly.

The issue with newly received spikes is accounting accurately for the delay. New spikes need to have the current time-step's time added; rowlets intended for the next time-step need an additional 1 added to the timer, since they are notionally being added one time-step later.

5.5 Instrumentation

It will be extremely useful to record the utilisation of the time budget of $20,000$ clock cycles per 0.1ms time-step to see where this budget is actually expended in particular instances of user-code. If more time is spent handling synapses, then it would make sense to expand the loop-unrolling of synapses at the expense of un-rolled loops of neurons. And *vice versa*, of course, if neuron code should be used more extensively than naïvely predicted.

An obvious and reasonably sensible approach might be to use recordings of previous runs to inform the compiler about which trade-offs to take. A particularly keen or brave (in the "Yes Minister!" sense) implementor might even think about changing the code on-the-fly (shudder).

Chapter 6

Superfast Strings: Speeding up String and Print Handling

Streamlining SpiNNaker, Part VI

Christian's work on `printf` and host-based string manipulation.