Contents

SIMD-oriented Fast Mersenne Twister (SFMT) implementation issues

Tiny Mersenne Twister (TinyMT) on pure Erlang and with NIFs
  • Implementation issues
  • Performance evaluation

Conclusions and future works
Disclaimer: non-goals

This presentation is NOT about cryptographically-secure RNGs

- Use **crypto** module functions (i.e., OpenSSL code) for **secure random number generation** in Erlang whenever available

In this presentation, PRNGs are:

- Of statistically uniform distribution
- Of predictable output with the same set of generation parameters and internal state
Why new PRNG for Erlang?

Speed
• Faster PRNG needed for simulations

Length of period
• Long enough to ensure statistical randomness

Size of internal state
• Memory footprint should be small for speed

Concurrent/parallel generation
• Mathematically-proven independent PRN streams should be capable
SFMT on Erlang (2011 Workshop)

SFMT PRNG characteristics

• Much longer period
  Typical length: \(2^{19937} - 1 \approx 10^{6002}\)
  (internal state: 156 128-bit words = 2496 bytes)
  OTP random module: \(\approx 7 \times 10^{12}\)
  (internal state: three 15-bit ints)

• BSD licensed (commercial use is OK)

• Implementation available in many languages
  Now default PRNG for Python and R
Our implementation: *sfmt-erlang*

We evaluated various period lengths

- We concluded $2^{19937}-1$ is optimal

**SFMT NIF: x3~4 faster than random**

- Compared time for each 32-bit generation
- SFMT generates random numbers of the internal table size at once
  
  Generation time is proportional to the table length

Now PropEr has a **building option** for sfmt-erlang (thanks!)
SFMT Implementation issues

Internal state size is large
• The state table should be placed in the Erlang BEAM shared heap

Slow without NIFs, but...
• Need for pure (non-NIF) working code exists
• NIFs may introduce instability

Generation of independent streams cannot be mathematically guaranteed
• Per-request creation of SFMT generation parameters is practically too slow
So what's new about TinyMT?

Iterative generation of 32/64-bit output
- Not like SFMT which generates as a batch

Period is shorter \((2^{127} - 1)\)
- It is still long enough for most simulation use

Small memory footprint (28 bytes)
- 127 bits for the internal state
- 3 x 32bit integers for the generation parameters

Independent stream generation capability
- \~2^{58} sets of the generation parameters
Our implementation: tinymt-erlang

Running fast enough in pure Erlang

- TinyMT has more complex algorithm than that of the `random` module, but has no floating-point calculation, so could be fast enough to compete with

Readability first, optimization second

- No tricky micro-optimization

Full compatibility with the `random` module

- Direct replacement for the existing code
Our assumptions on TinyMT

Smaller state size = faster speed
• TinyMT: 28bytes, SFMT: 2496bytes

Simpler algorithm = faster in pure Erlang
• TinyMT: 2 functions, 106 lines of 'S' file
• SFMT: 5 functions, 503 lines of 'S' file
('S' = R15B01 BEAM assembly source file)

Easier generation of independent streams from the same algorithms
• It's possible on SFMT too, but needs a lot of pre-computation and cannot be changed as needed
Design details of **tinytm-erlang**

32-bit output implementation only
No floating point calculation
No Erlang case statement

Note: Erlang integers are BIGNUMs

- Bitmasking by `band` operator needed for implementing C-like integer operations

State in a single record `#intstate32{}`

- Internal state and the generation parameters can be separately modified
Design tips for [1,N] integer RNGs

Equivalent to [0, N-1] integer RNGs

Ensuring equal probabilities

• Multiplication of [0, 1) floating-point numbers to generate integer numbers may cause errors unless N = 2^n (order of error: N x 2^(-32))

• A right way (implemented in `tinymt-erlang`)
  A) Generate a 32-bit integer random number R
  B) Compute Q which is the closest multiple of N to 2^32 (Q rem N =:= 0, 0 =< (2^32 - Q) =< (N - 1))
  C) If R > Q, try the generation at A) again; else compute the result as (R rem N) + 1.
Our test environments

Erlang/OTP R15B01 on:

- Intel Xeon E5-2670 x 8 (16 cores), 2.6GHz clock, RedHat Enterprise Linux 6, x86_64
  KU ACCMS Supercomputer Cluster B batch node
- Intel Core i5-2410M (4 cores), 2.3GHz clock, FreeBSD/amd64 9.0-STABLE (a notebook PC)
- Intel Atom N270 (2 cores), 1.6GHz clock, FreeBSD/i386 8.3-RELEASE (a netbook PC)
Pure Erlang wall clock test results

TinyMT execution speed against the random module (wall clock):

- x86_64/amd64: x0.93~x1.21 (same speed)
- x86/i386: x0.31~x0.34 (much slower)

Speed gain of HiPE o3 option comparing to the non-HiPE version (wall clock):

- x86_64/amd64: $x2.4\sim x3.6$ (much faster)
- x86/i386: x1.25 (TinyMT is still slower (x0.4))
Pure Erlang fprof test results

By accumulated time measured by fprof:
TinyMT takes x2~x6 execution time than that with the random module

• For uniform_s/1 (float RNG): TinyMT has to do the integer-to-float conversion, while the random module generates the float result first

• For uniform_s/2 ([1, N] integer RNG): it is faster than uniform_s/1 on TinyMT, but still x0.5 speed (slower) than that of the random module
Observations from pure Erlang tests

Overhead to call functions is significant

- TinyMT needs two function calls to generate a result; aggregation to a NIF will be effective

Integer operation overhead

- 32bit unsigned integers are BIGNUMs on x86
  - Small Integers in a word: max 28bits
    - Operations will be much efficient in C than Erlang

NIFs for core functions will be effective

- For the functions called many times only
NIF fproff test results

uniform_s/1: x3 faster than non-NIF
  • the same speed as the random module
uniform_s/2: x7 faster than non-NIF
  • x3 speed as the random module

SFMT is x1.52 faster than TinyMT

1 million calls/second seems to be the upper limit for KU ACCMS nodes
Observation from NIF tests

Our assumptions **proven false:**

- Internal state size is **irrelevant** to speed which may differ in a memory-constrained system
- Simpler algorithm does **not necessarily** mean faster

Batch generation of multiple numbers is essential for gaining speed

- Overhead of calling functions and BEAM memory allocation are presumably significant
Even more NIF tests (latest work)

After the paper submission, batch list generation NIFs are added

- Speed gain: x6~7 on wall clock, x13 on fprof comparing to the non-batch NIFs
- C compiler inline optimization (as in the sfmt-erlang) even makes the code x1.4 faster

sfmt-erlang batch generation is still x3~4 faster than tinymt-erlang in fprof

- More optimization needed

  On memory allocation with enif_*() functions
## NIFs and BEAM scheduling issue

Which is better to avoid scheduler hiccups due to the NIFs occupying the CPU time?

* On `sfmt-erlang` and `tinymt-erlang` batch NIFs

<table>
<thead>
<tr>
<th>High workload of batch processing (list generation)</th>
<th>Low workload of copying the results</th>
</tr>
</thead>
</table>

* On `tinymt-erlang` per-request NIFs and pure Erlang code

| Continuous not-so-high workload of per-request NIF processing and instructions executed by BEAM |
Related works

TinyMT key pre-computation
• Took 32 days to generate $2^{28}$ sets of 32/64-bit TinyMT generation parameters
• $18\sim19$ keys/sec for each CPU of KU ACCMS cluster

SFMT and TinyMT seed jumping
• Fast computation of multiple state transitions
• Useful for multiple independent generation

Wichmann-Hill 2006
• Successor of the algorithm of random module
• Output independency of proposed seeding for parallel generation is not firmly proven as in TinyMT
• Licensing issues exists (non-open license)
• Michael Truog built the BIGNUM version
Conclusion

TinyMT is a viable candidate for replacing Erlang/OTP stock non-secure PRNG

- The pure Erlang code is fast enough especially with HiPE compilation
- By NIFnization, the speed is the same as the stock random module
- When using batch generation NIFs, the speed is x7 faster, though sfmt-erlang is still x3~4 faster than tinymt-erlang
Future works

Exploring more parallelism

• Use case proof of multiple independent stream generation is essential
• Distribution schemes for key generation parameters is essential
  e.g., through message queues

More optimization needed

• More performance improvement is possible
  Memory allocation strategy of NIFs
Questions?