SFMT Pseudo Random Number Generator for Erlang

Kenji Rikitake

ACCMS/IIMC,

Kyoto University

23-SEP-2011

Contents

RNGs implemented in Erlang/OTP

- crypto and random modules and their issues

SIMD-oriented Fast Mersenne Twister (SFMT) on pure Erlang and with NIFs

- implementation issues
- performance evaluation

Conclusions and future works

RNG libraries in Erlang/OTP

crypto module: OpenSSL API

- rand_bytes/1, rand_uniform/2
- strong_rand_bytes/1, strong_rand_mpint/3
- NIFs since R14B

random module: Wichmann-Hill AS183

- published in 1982 for 16-bit calculations
- period is very short (~ 7 x 10^12)
- Written solely in Erlang

random module code of AS183

```
%% from lib/stdlib/src/random.erl
%% of Erlang/OTP R14B02

uniform() ->
    {A1, A2, A3} = case get(random_seed) of
                       undefined -> seed0();
                       Tuple -> Tuple
                   end,
    B1 = (A1*171) rem 30269,
    B2 = (A2*172) rem 30307,
    B3 = (A3*170) rem 30323,
    put(random_seed, {B1,B2,B3}),
    R = A1/30269 + A2/30307 + A3/30323,
    R - trunc(R).
```

Issues needed to be solved

Source code audit needed

- Erlang ssh module session key revelation
  - Fixed on R14B03, US-CERT VU#178990
  - In ssh module of R14B02 only AS183 was used

Longer period for non-crypto RNGs

- AS183 is good, but we need something better
  - only holds ~81 days for 1 million numbers/sec

Faster generation for non-crypto RNGs

- Faster algorithm for integer use
- Maybe even faster with NIFs

SIMD-Oriented Mersenne Twister

A very good and fast PRNG

- very good = very long period

  typical: 2^19937 - 1, up to 2^216091 – 1

  (depending on the internal state table size)
- Open source and (new) BSD licensed
- Implementations avaliable for:

  C, C++, Java, Python, R, etc.
- Supporting SSE2/altivec SIMD features

SFMT Step 1: reentrant C code

Revised the SFMT reference code

- Removed all `static` arrays

    The internal state table was defined as `static`

        the ultimate form of **the shared memory evil!**

- Rewritten the code so that the internal state tables must be passed by the pointers

    Allowing concurrent operation of the functions

- Removed the altivec and 64bit features

    no testing environment available

SFMT Step 2: pure Erlang version

SFMT itself can be written as a recursion

```
a[X]=r(a[X-N],a[X-(N-POS1)],a[X-1],a[X-2])
```

Extensive use of head-and-tail list

- Making lists for each recursive argument
- list head addition + lists:reverse/1 is 50% faster than using the ++ (append) operator
- See Figs. 1 and 3 of the paper for the details

Still ~300 times slower than the C Code

C to Erlang conversion tips

Erlang integers are **BIGNUMs**

- Bitmasking by `band` needed for C-like integers
  whenever overflow is possible (e.g., `bsl`)

Erlang `bsr` is **arithmetic shift right**

- e.g., `-1 =:= -10 bsr 4` is `true`
- For **logical** shift right, **use positive numbers**

Avoid arrays; lists are cheaper

- The array module object is **immutable**
  i.e., array:set/3 makes a modified copy

SFMT Step 3: NIF programming

NIFs are C **static** code, shared-everything

- One `.c` file per each module
- Make the copy first before modifying a binary

  Without this you may face a heisenbug

  Erlang binaries are supposed to be **immutable; so the content must stay unmodified!**
- Study the erl_nif manual under erts

  Learn `enif_*()` functions for efficient codin

  R14 crypto module is another good example

A case study: table handling on SFMT

Case 1: list processing

- NIF: internal table -> integer list
- generating PRN by `[head|tail]` operation

Case 2: random access through NIF

- generating PRN each time by calling a NIF with the internal table and the index number

Result: Case 1 is faster than Case 2

- on a 2-core SMP VM - parallelism discovered?
- Lesson learned: **profile before optimize**

So how fast the SFMT NIF code is?

Wall clock time of 100 * 100000 PRNs

- When n=19937, N=156
- on Kyoto University ACCMS Supercomputer Thin Cluster node (Fujitsu HX600)

  AMD Opteron 2.3GHz amd64 16 cores/node

  RedHat Enterprise Linux AS V4

  Erlang R14B03, running in a batch queue

sfmt: gen_rand_list32/2	sfmt: uniform_s/1	random: uniform_s/1	sfmt: gen_rand32_max/2	random: uniform_s/2
240ms	2430ms	7680ms	2440ms	8310ms
x1.0	**x10.1**	**x32.0**	**x10.2**	**x34.6**

Total exec time of sfmt:gen_rand32_max .vs. SFMT internal table length

(for 100 * 100000 calls)

+ leciel
× reseaux
△ thin

Total execution wall clock time [ms]

8000
6000
4000
2000
0

0
500
1000
1500

N (internal table length [of 128bit words]) for N = [5, 34, 156, 674, 1689]

SFMT gen_rand32_list/2 performance

Total OWN time measured by fprof for 10 calls of gen_rand_list32(10000, State) of each sfmt module

total OWN time [ms]

0
2
4
6
8

+ leciel
× reseaux
△ thin

0
500
1000
1500

N (internal table length [of 128bit words]) for N = [5, 34, 156, 674, 1689]

SFMT gen_rand_all/1 performance

gen_rand_all/1 OWN time measured by fprof, for 100000 integer and 100000 float random numbers of sfmt modules measured on thin (Kyoto University ACCMS supercomputer)

OWN time for each call [microseconds]

0
50
100
150
200

0
500
1000
1500

+ leciel
× reseaux
△ thin

N (internal table length [of 128bit words]) for N = [5, 34, 156, 674, 1689]

Summary of the performance evaluation

SFMT NIF: x3~4 faster than random

High exec overhead for shorter period

- especially for 2^607 - 1

Exec own time of list generation are same

Exec own time per NIF call to update the internal table is proportional to the length

These characteristics remain the same between three different exec environments

Related Works

Wichmann-Hill 2006 (random_wh06)

- Period: ~2^120, state: 4 x 31-bit integers
- Key-generation support of independent streams
- Pure Erlang version tested at EF SF Bay 2011
- Exec time increase from random module:

  <10% with sufficient floating-point support

  >60% on Atom (lesser capability)

Tiny Mersenne Twister (TinyMT)

- Period: 2^127 - 1, state: 127bits
- Jump function for n-th fast-forward calculation

  Generation of discrete sub-sequences possible
- Evaluation for Erlang ongoing

Conclusion and future works

SFMT with NIFs is pratical on Erlang

- State generation may affect the scheduling

Future works: exploring parallelism

- Multiple streams as sub-sequences
- New algorithms needed for faster speed and smaller memory footprint
  - SFMT looks too heavy-weight for parallelism
  - Candidates: Wichman-Hill 2006, TinyMT