Implementing SFMT PRNG on Erlang

Kenji Rikitake

Academic Center for Computing and Media Studies (ACCMS),

Kyoto University

27-AUG-2010

My contribution to SFMT on Erlang

Making the C code reentrant
- http://github.com/jj1bdx/sfmt-extstate

Writing a pure Erlang version of SFMT

Writing a NIF version of SFMT
- with the reentrant C code
- ~40 times faster than the pure Erlang code
  - it's even faster than random module

Available in Github at
- http://github.com/jj1bdx/sfmt-erlang

Kyoto University

Pseudo-Random Number Generator

PRNG is:

- (predictable) random number sources
- a sequence generated from an internal state

  {NewRN, NewState} = prng_fun(CurrentState)

You need a PRNG for:

- application testing (random sampling)
- generating artificial noise signals
- random hashing

  hash tables, TCP/UDP source port selection, etc.

PRNG library in stock Erlang

stdlib random module

- an old alrogithm made in 1982[1]

  Cycle length is very short (~ 7 x 10^12) [2]

  a revised alrogithm already published [2]

- Written in pure Erlang, but…

  Some functions use the process dictionaries

  (for gaining the speed?)

[1] B. A. Wichmann, I. D. Hill, “Algorithm AS 183: An Efficient and Portable Pseudo-Random Number Generator”, Journal of the Royal Statistical Society. Series C (Applied Statistics), Vol. 31, No. 2 (1982), pp. 188-190, Stable URL: http://www.jstor.org/stable/2347988

[2] B.A. Wichmann, I.D. Hill, Generating good pseudo-random numbers, Computational Statistics & Data Analysis, Volume 51, Issue 3, 1 December 2006, Pages 1614-1622, ISSN 0167-9473, DOI: 10.1016/j.csda.2006.05.019.

SIMD-Oriented Mersenne Twister

A very good and fast PRNG

- A revised version of Mersenne Twister
- very good = very long generation cycle
  typical: 2^19937 - 1, up to 2^616091 – 1
  (depending on the internal state table size)
- Supporting SSE2/altivec SIMD features
- Open source and (new) BSD licensed
- Implementations of (SF)MT avaliable for:
  C, C++, Gauche, Java, Python, R, etc.

URL: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html

So why SFMT on Erlang?

The PRNG quality is well proven

- survived the DIEHARD test

It would be fast if implemented with NIFs

- and that's what I have done

SFMT RNG parameters are tunable

- multiple algorithms generating independent streams possible if needed

A better PRNG is needed for security

- DNS UDP port number exploitation attack

Step 1: making the C code reentrant

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!**

- Removed the altivec and 64bit features

  no testing environment available

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

  Allowing concurrent operation of the functions

Step 2: writing a pure Erlang SFMT

Literal translation from the revised C code

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 lists

- Adding elements to the heads and do the lists:reverse/1 made the code 50% faster than using the ++ operator

It was ~300 times slower than the C Code

- But it worked! (And that's what is important)

C to Erlang conversion tips

Erlang integers are **BIGNUMs**

- Explicitly limit the result bit length by `band` each time after `bsl` and any other operation which may exceed the given C integer length

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

- e.g., the value of `-1 =:= -10 bsr 4` is `true`

The array module can emulate C arrays

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

Kyoto University

Step 3: writing a NIF version of SFMT

NIF modules are full of C `static` code

- It's a shared-everything world as default
- When a NIF fails, it crashes the BEAM

The fastest way to learn the NIF coding:

- read the manual of erl_nif (under erts)
- read the R14 crypto module
- try first from smaller functions, step-by-step
- prepare regression testing code (e.g., eunit)

Kyoto University

NIF programming tips

It's hard-core C programming

- Put all functions in the same `.c` file
  - Do you remember how the `static` scope work?
- Make the copy first before touching a binary
  - Without this you may face a heisenbug
  - Erlang binaries are supposed to be **immutable; so the content must be left untouched!**
- Learn the `enif_*()` functions first
  - they will make the code efficient and terse

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**

For the efficient Erlang + C coding

Use dev tools as much as possible

- rebar is an excellent autobuilding tool
  - http://hg.basho.com/rebar
- EUnit is well-suited for functional unit testing
  - available in the Erlang/OTP distribution

Automate the documentation

- EDoc and Doxygen help me a lot
- Learn the Markdown format
  - It's much easier than to write HTML by hand

So how fast the code is?

Wall clock time of 100 * 100000 PRNs

- on Kyoto University ACCMS Supercomputer Thin Cluster node (Fujitsu HX600)
  - AMD Opteron 2.3GHz amd64 16 cores/node
  - RedHat Enterprise Linux AS V4

sfmt: gen_rand_list32/2	sfmt: uniform_s/1	random: uniform_s/1	sfmt: gen_rand32_max/2	random: uniform_s/2
270ms	2550ms	7100ms	2390ms	7560ms
x1.0	x9.4	x26.3	x8.9	x28.0

Thoughts on the results

Erlang code: still ~x10 slower than C

- SSE2 code crashes for an unknown reason
  - 128-bit alignment issue of `enif_alloc()`?

sfmt NIF: >x3 faster than random module

Future works: exploring parallelism

- SFMT has the sequential characteristics
- Looking for a new algorithm is needed
  - There are parallelism-oriented PRNG algorithms

Kyoto University

Acknowledgments

ACCMS, Kyoto University (my employer)

- For financially supporting the work
- In this research, I used the Kyoto University ACCMS Supercomputer Thin Cluster System
  - It's more cost effective than building an amd64 test environment on an independent PC

Those who helped the development:

- Dave "dizzyd" Smith, Tuncer Ayaz, Tim Bates, and Dan Gudmudsson

Kyoto University