Erlang/OTP and how the PRNGs work

Kenji Rikitake

Academic Center for Computing and Media Studies (ACCMS),

Kyoto University

25-MAR-2011

Contents

What is random number generator?

Requirement of RNGs

- True RNGs .vs. Pseudo RNGs (PRNGs)

RNGs implemented in Erlang/OTP

- crypto and random modules and their issues

PRNG enhancements

- SFMT: a long-period PRNG with NIFs
- Wichmann-Hill 2006 algorithm (random_wh06)

Conclusions and future works

What is random number generator?

Generating sequence of discrete numbers

Two types of RNGs:

- "True" RNGs: data from physical phenomena
- Pseudo RNGs: computed from a seed

    seed: initial vectors of tables of the internal state

In Erlang/OTP, two modules of RNGs

- crypto: OpenSSL API (NIFs from R14B)
- random: Wichmann-Hill AS183 (in 1982)

Requirements of RNGs

Uniform deviates

- Each of possible values is **equally probable**
- The building block for other deviates

Each number in the sequence must be **statistically independent**

- Non-deterministic (unpredictable from past)
- Non-periodic (no same sequence reappears)

Fast enough to supply the demand

- Generation speed could be a bottleneck

"True" RNG hardware examples

Collecting physical randomness / entropy

- Avalanche diode noise
- Free-running oscillators
- Atmospheric noise (random.org uses this)

Slow and expensive

- The generation process does **not** guarantee if the output is equally probable and statistically independent
- The output should be continuously verified and calibrated if the offset of the output deviate is large

See RFC4086 Section 3 and Section 4 for the details

Not repeatable (at least theoretically)

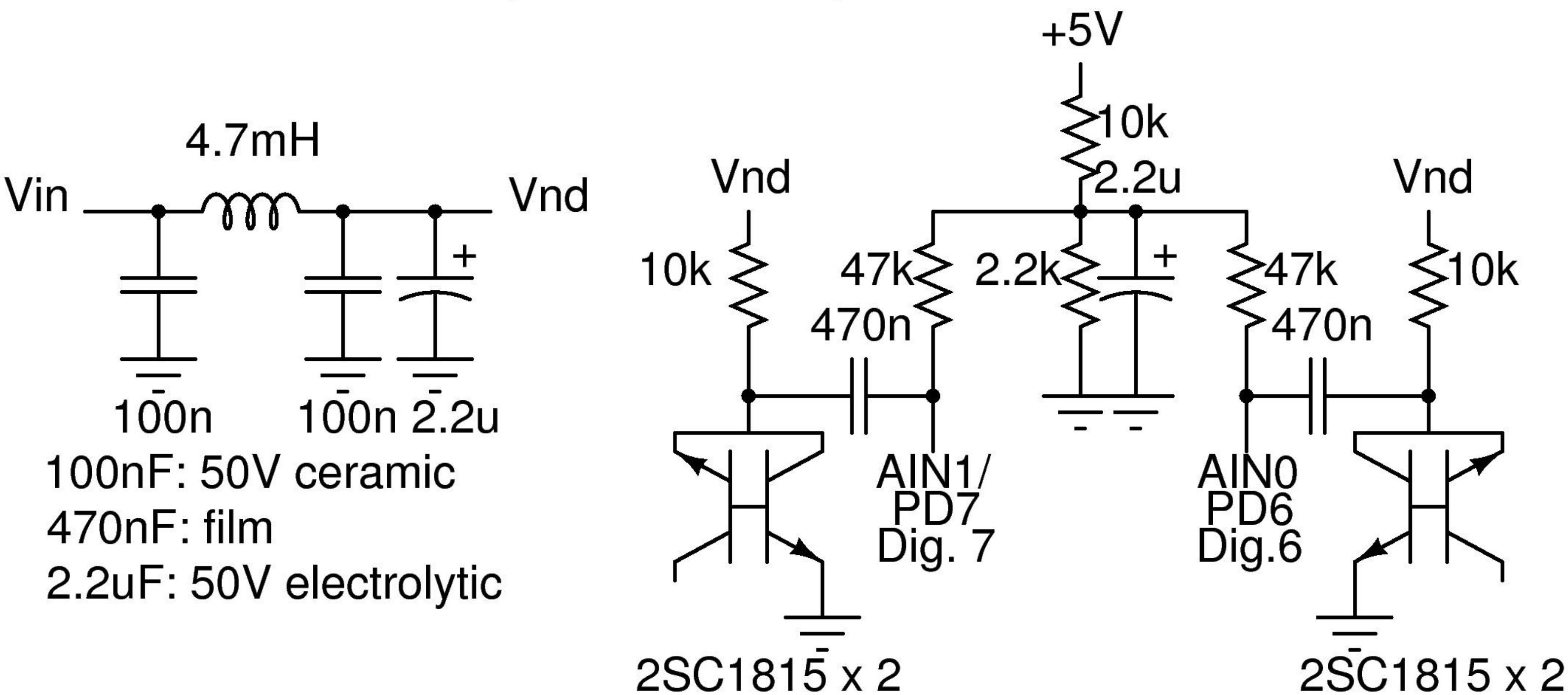
- Practically used for seeding PRNGs for cryptography

Avalanche diode RNG circuit example

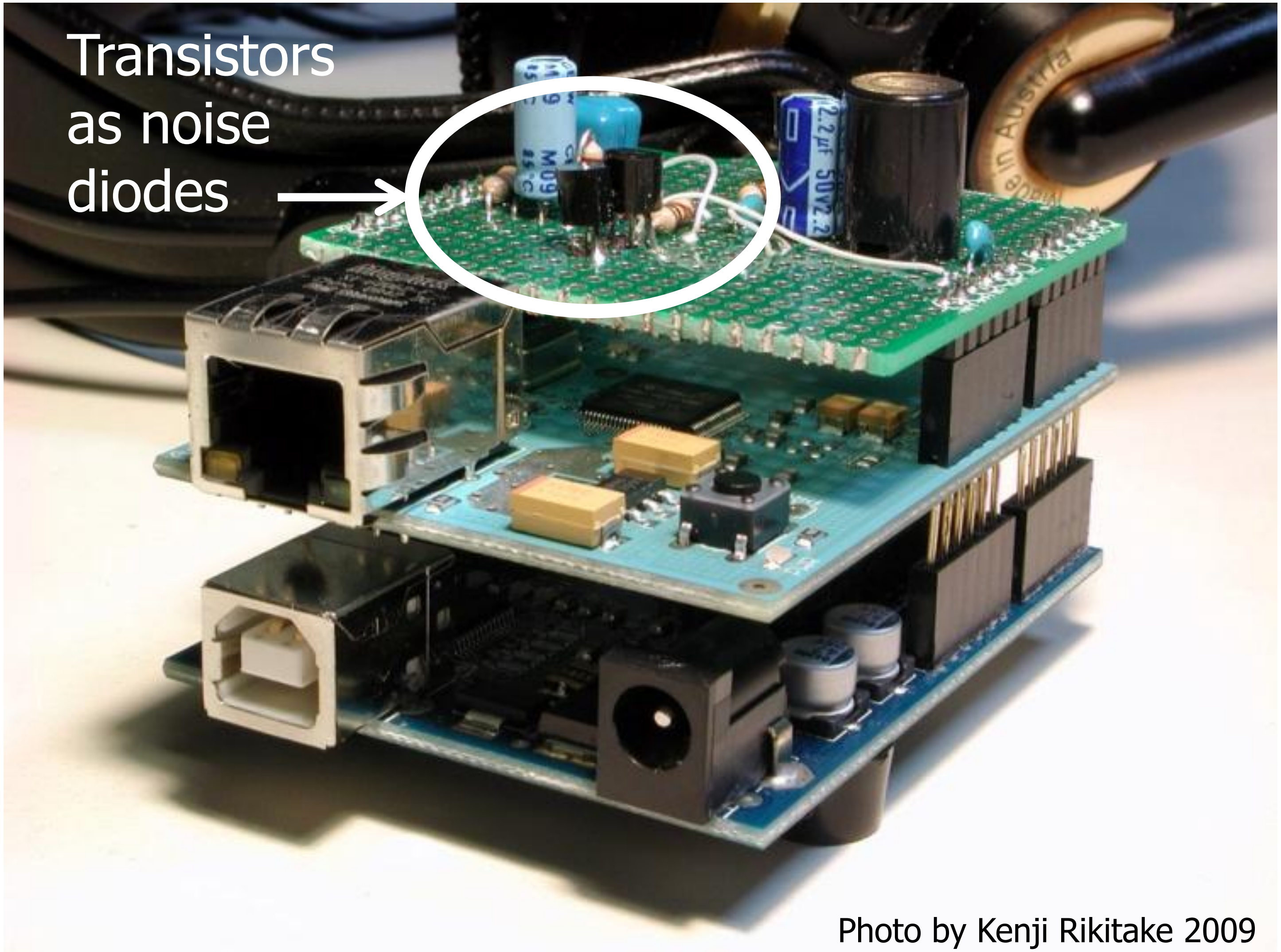
Example at https://github.com/jj1bdx/avrhwrng/
Speed: ~10kbps (or even slower for accuracy)

Arduino Duemilanove shield schematics
for a hardware random number generator
by Kenji Rikitake 6-APR-2009

Vin = +12V or +13.8V (+9V didn't work)



Arduino RNG looks like this



Characteristics of pseudo RNGs

Computed number sequences

- Deterministic by definition
  - given the same seed, the same results show up
- Very long period but periodic anyway
  - Longer period needed for larger scale application
- Faster and more efficient than "True" RNGs

Practical use: simulation and modeling

- random sampling / hashing / testing
  - Load balancing, DHT, Monte Carlo method, etc.

Cryptographic strength of PRNGs

Cryptographically-strong PRNGs must:

- use the algorithm to prevent future data from the past generated data (with AES, SHA, etc.)
- maintain collection of entropy pools from the various sources (network activities, etc.)

  virtual machines: less entropy will be obtainable
- secure the seeding process to prevent injection attempts from the attackers

Use well-established methods for security

- OpenSSL uses /dev/urandom on FreeBSD
- Accuracy transcends speed

  Expect a lot of time to obtain sufficient random bits

So what kind of RNGs in Erlang/OTP?

crypto module

- rand_bytes/1, rand_uniform/2

  OpenSSL API functions

- Always use crypto functions for security

random module: Wichmann-Hill AS183

- period is very short (~ 7 x 10^12) [1]
- Written solely in Erlang

[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

Original AS183 code in FORTRAN

```
C IX, IY, IZ SHOULD BE SET TO INTEGER VALUES
C BETWEEN 1 AND 30000 BEFORE FIRST ENTRY

IX = MOD(171 * IX, 30269)
IY = MOD(172 * IY, 30307)
IZ = MOD(170 * IZ, 30323)

RANDOM = AMOD(FLOAT(IX) / 30269.0 +
         FLOAT(IY) / 30307.0 + FLOAT(IZ) /
         30323.0, 1.0)
```

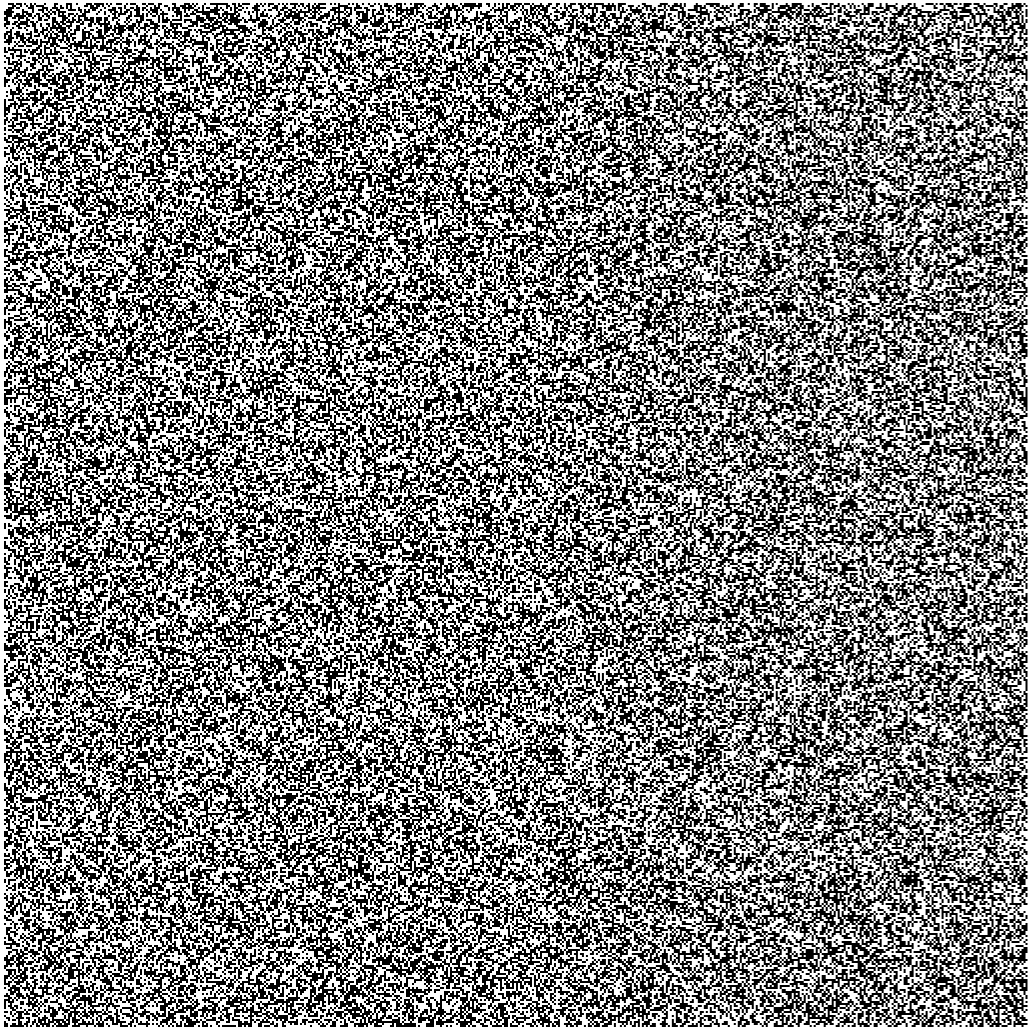
Source: Microsoft, Description of the RAND function in Excel
http://support.microsoft.com/kb/828795

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).
```

AS183 512x512 bitmap pattern test



(this looks well-randomized visually)

What weak or bad RNGs will cause

Vulnerability by predictable choice

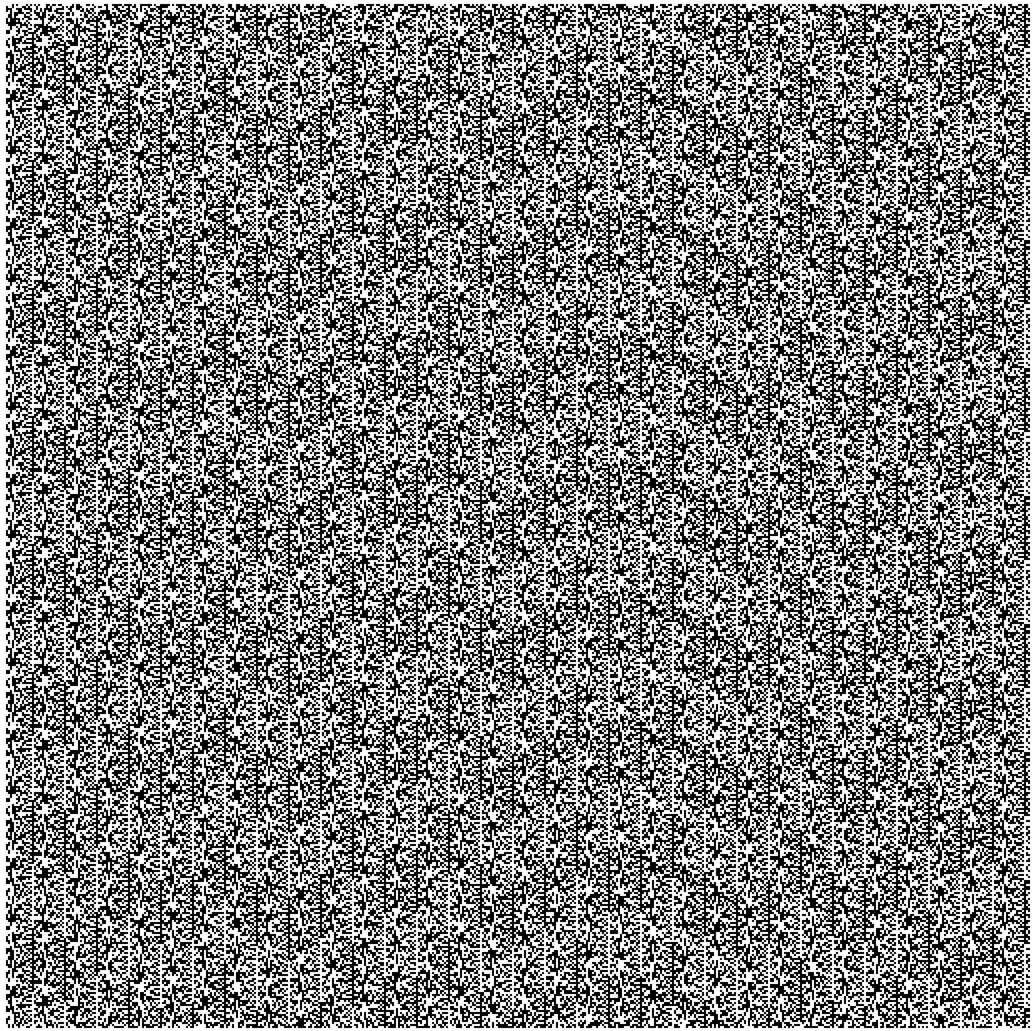
- DNS UDP source port numbers
- Precisely guessing cross-site state through JavaScript Math.random() method [2]

Non-uniform bias on simulation

- Which may show up on a short-period RNG
- Assumption of uniform deviate may fail

[2] A. Klein: Temporary user tracking in major browsers and Cross-domain information leakage and attacks, Trusteer, 2008, URL: http://www.trusteer.com/list-context/publications/temporary-user-tracking-major-browsers-and-cross-domain-information-leakag

rand(0,1) on PHP 5 Windows



Source: http://twitpic.com/gq81b/full

(you can see a repetitive pattern - that's bad)

Another popular example of bad RNG

```
%% originally from http://xkcd.com/221/
%% converted(?) to Erlang by Kenji Rikitake

-module(get_random_number).
-export([rand/0]).

rand() ->
    % Chosen by fair dice roll.
    % Guaranteed to be random.
    4.

%% DO NOT USE THIS FOR A REAL APPLICATION!
```

Issues needed to be solved

For security, crypto functions are must

- In ssh module of R14B02 only AS183 found

Longer period for non-crypto RNGs

- AS183 is good, but we need something better

  7 x 10^12 period only holds ~81 days, if you generate 1 million random numbers for each second

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

- A revised version of Mersenne Twister
- very good = very long generation cycle

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

PRNG enhancements with sfmt-erlang

SFMT implementation

- Making the C code reentrant
  http://github.com/jj1bdx/sfmt-extstate
- of five (5) different periods with NIFs
  ~40 times faster than the non-NIF code
  it's even faster than random module

Wichmann-Hill 2006 generator [3]

- Called random_wh06 module
- A better RNG when NIFs can't be used

[3] 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.

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!**
- Removed the altivec and 64bit features
  - no testing environment available
- SSE2 code removed
  - crashes for an unknown reason
    - 128-bit alignment issue of `enif_alloc()`?
- Rewritten the code so that the internal state tables must be passed by the pointers
  - Allowing concurrent operation of the functions

SFMT Step 2: pure Erlang version

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 head-and-tail lists

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

Still ~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., `-1 =:= -10 bsr 4` is `true`
- The array module object is **immutable**
  - i.e., array:set/3 makes a modified copy

SFMT Step 3: writing a NIF version

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
- Use regression testing tools (e.g., eunit)

NIF programming tips

It's hard-core C programming

- Put all functions in the same `.c` file

  Remember how `static` scope works

- 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!**

- 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 a decent syntax highlighter

- erlang-mode and cc-mode on Emacs

Use dev tools as much as possible

- eunit, fprof, rebar, escript, etc.

Automate the documentation

- EDoc (for Erlang) and Doxygen (for C)
- Learn the Markdown format
  - It's much easier than to write HTML by hand

So how fast the SFMT NIF 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
  - Erlang R14B01, running in a batch queue

sfmt: gen_rand_list32/2	sfmt: uniform_s/1	random: uniform_s/1	random_wh06: uniform_s/1	sfmt: gen_rand32_max/2	random: uniform_s/2	random_wh06: uniform_s/2
240ms	2600ms	7110ms	11220ms	2440ms	7720ms	11790ms
x1.0	**x10.8**	**x29.6**	**x46.8**	**x10.2**	**x32.2**	**x49.1**

speed of random .vs. random_wh06

For 100000 calls of OWN time measured by fprof on R14B01

System details:

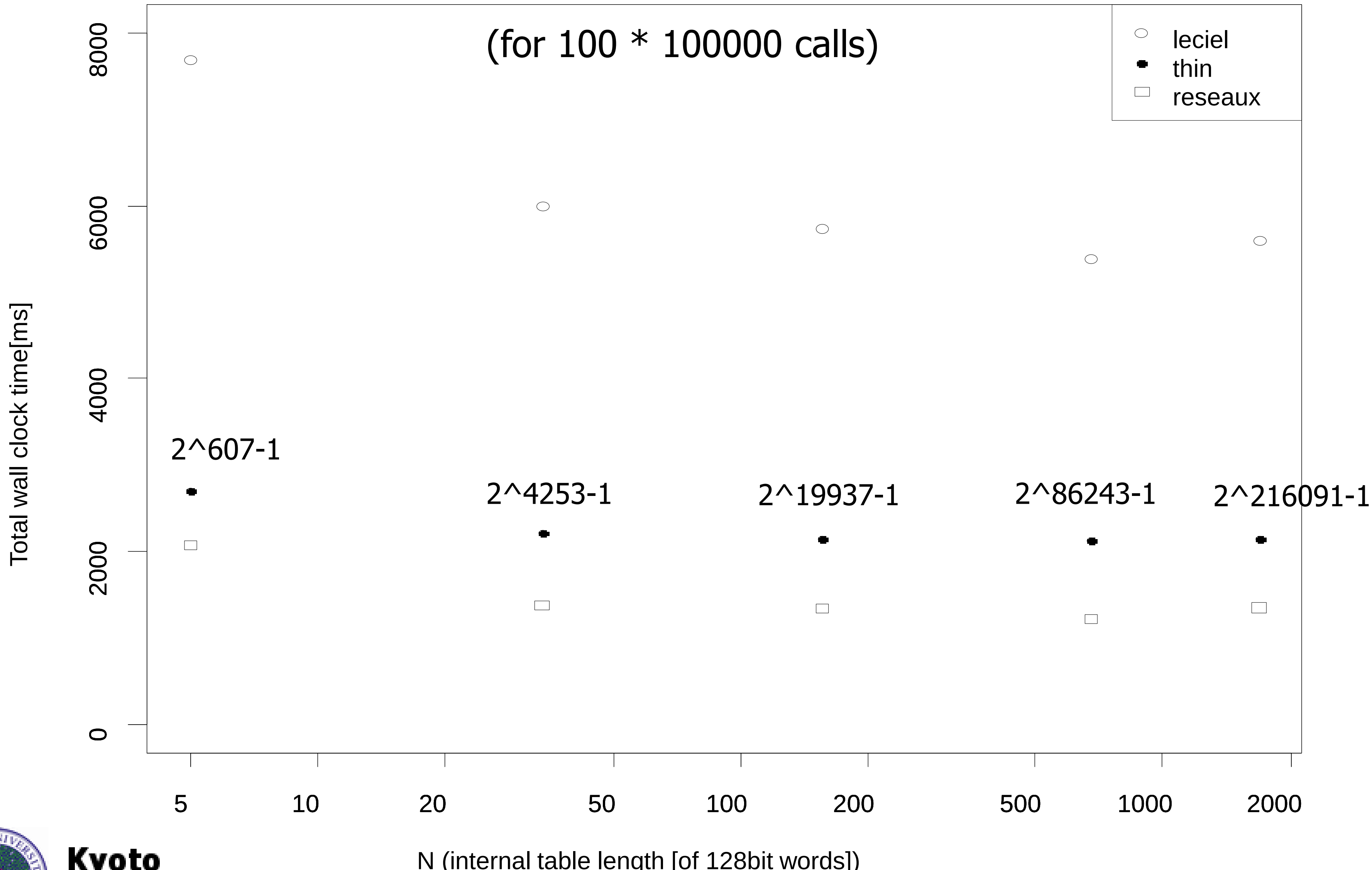
- reseaux: Core2Duo E6550 2.3GHz FreeBSD/i386 8.2-RELEASE
- leciel: Atom N270 1.6GHz FreeBSD/i386 8.2-RELEASE
- thin: Opteron 8356 2.3GHz RHEL AS V4 on amd64

This set of results suggest:

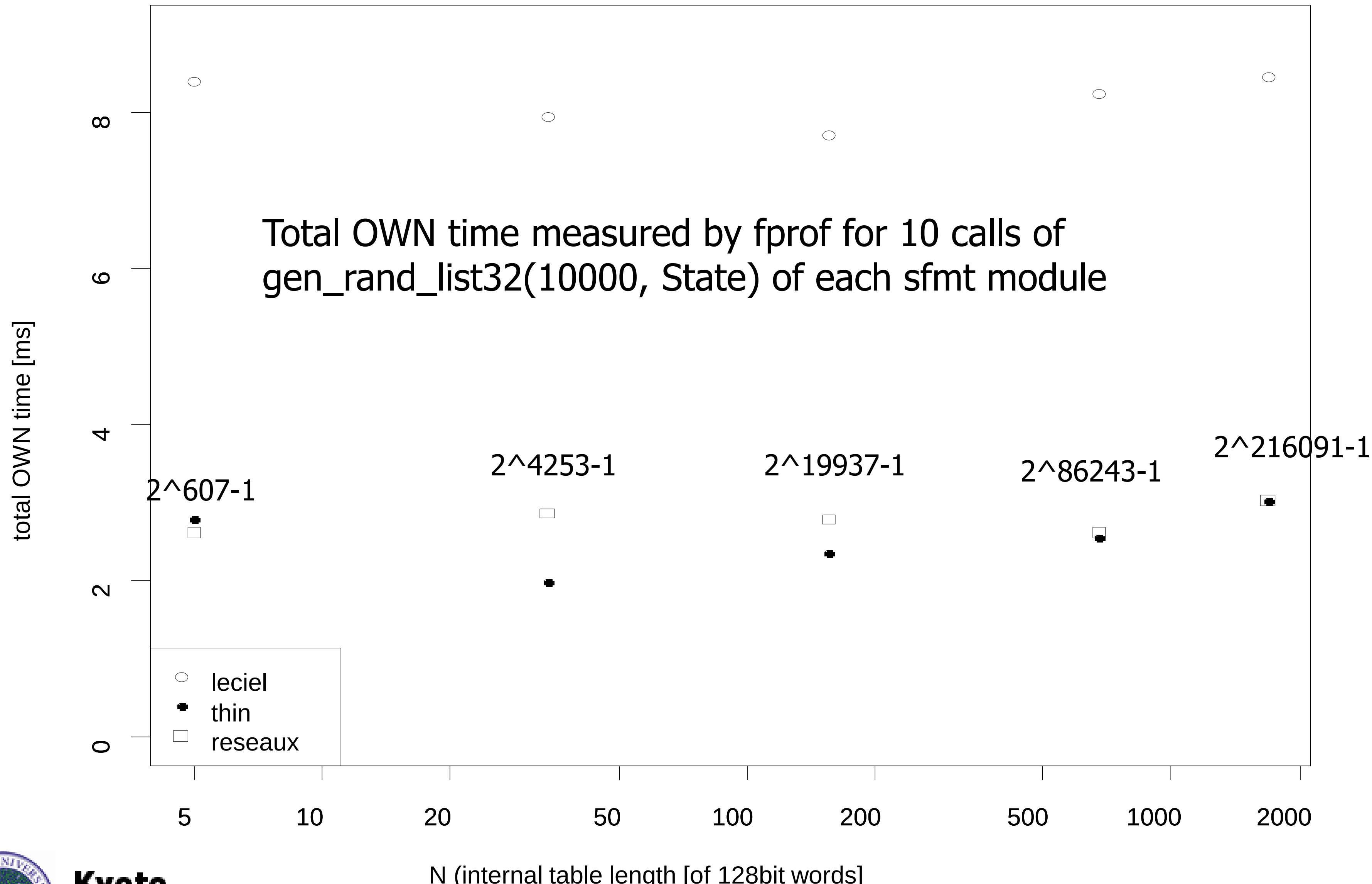
- The speed overhead from random to random_wh06 for CPUs with sufficient floating-point calculation support: < 10%
- On a CPU with lesser capability such as Atom, the overhead will increase to > 60%

	random:uniform_s/1	**random_wh06:uniform_s/1**	**ratio of random_wh06 / random**
reseaux	544.9ms	487.9ms	0.895
leciel	1400.3ms	2274.8ms	1.625
thin	309.2ms	331.2ms	1.071

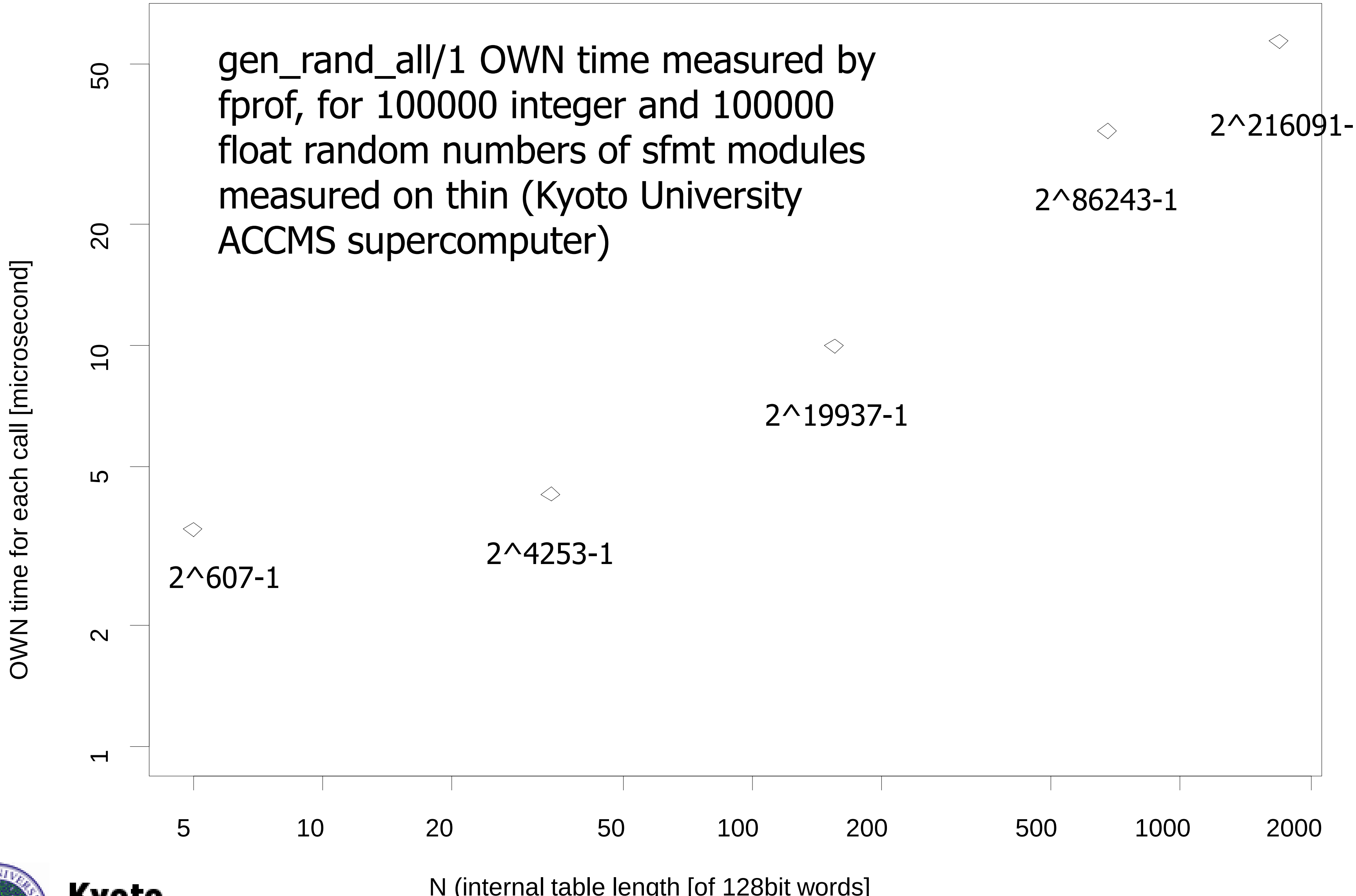
Total exec time of sfmt:gen_rand32_max .vs. SFMT internal table length



SFMT gen_rand32_list/2 performance



SFMT gen_rand_all/1 performance



Conclusion and future works (1)

SFMT NIF: >x3 faster than AS183

- It's also better for simulation and modeling

SFMT NIF behavior for period length

- Shorter period causes larger calling overhead
- gen_rand32_list/2 exec time is ~ constant
- gen_rand_all/1 exec time is proportional to the internal state table size for a large period

random_wh06: 10~60% slower than AS183

- more room to optimize for slower CPUs
  - Full 32bit integer is BIGNUM for 32bit Erlang VM

Conclusion and future works (2)

Future works: exploring parallelism

- SFMT is inherently sequential/iterative
- Looking for a new algorithm is needed
  - There are parallelism-oriented PRNG algorithms
  - Simplistic algorithms: LShift, XOR32, etc.

Review of Erlang/OTP code for the secure usage of PRNGs is needed

- Very few network modules use crypto RNG
- Analysis on Windows and other OSes needed

Acknowledgments to:

ACCMS, Kyoto University

- 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

People helping the code development:

- Dave "dizzyd" Smith, Tuncer Ayaz, Tim Bates, Dan Gudmudsson, Richard O'Keefe

and all the participants of EF SF Bay 2011!

References

- https://github.com/jj1bdx/sfmt-erlang/
- random.org http://www.random.org/
- Press et al, Numerical Recipes (Third Edition), Cambridge Press, 2007, ISBN 9780521880688, Chapter 7 "Random Numbers", see http://www.nr.com/
- http://www.diigo.com/user/jj1bdx/random
  - My bookmarks about random number generation
- Ferguson et al, Cryptography Engineering, Wiley, 2010, ISBN 9780470474242 , Chapter 9 "Generating Randomness"