Fast and robust PRNGs based on jumps in N-cubes for simulation, but not exclusively for that.

Sylvain Contassot-Vivier, Jean-François Couchot, Mohammed Bakiri, Pierre-Cyrille Heam

Fast and robust PRNGs based on jumps in N-cubes for simulation, but not exclusively for that.

Sylvain Contassot-Vivier*, Jean-François Couchot*†, Mohammed Bakiri‡, and Pierre-Cyrille Héam†
* LORIA, CNRS, University of Lorraine, France
Email: Sylvain.Contassotvivier@loria.fr
† FEMTO-ST Institute, University Bourgogne Franche-Comté, France
‡ Development Center of Advanced Technologies, Algeria

***Abstract*—Pseudo-Random Number Generators (PRNG) are omnipresent in computer science: they are embedded in all approaches of numerical simulation (for exhaustiveness), optimization (to discover new solutions), testing (to detect bugs) cryptography (to generate keys), and deep learning (for initialization, to allow generalizations).... PRNGs can be basically divided in two main categories: fast ones, robust ones. The former have often statistical biases such as not being uniformly distributed in all dimensions, having a too short period of time,.... In the latter case, statistical quality is present but the generators are not fast. This is typically what is encountered when running a cryptographically secure PRNG. In this paper, we propose alternative architectures, based on jumps in N-cubes, that provide fast and robust PRNGs for efficient simulations, but not exclusively for that.**

***Index Terms*—PRNG, Hamiltonian Cycles, FPGA, Simulation.**

I. Introduction

Hardware devices for performing simulations are multiple: multi-core processors, GPU cards, FPGA cards. Each device is dedicated to a family of simulations and none should be privileged over the others. Simulation methods, prediction of parameters such as the Monte-Carlo method, are based more or less on the generation of random events that allow the simulation of random physical events [1]. Many (pseudo)random number generators have been proposed, but few can be used for serious simulations: their statistical qualities are not sufficient to ensure that the simulation will reflect all the cases that may appear in reality.

But the use of Pseudo-Random Number Generators (PRNG) is not limited to simulation, even if this field represents an important part of their applications. PRNGs can indeed be found in public applications like random game universes, search for new optimums in operational research, cryptography, ...

Design of such PRNGs must fulfill some essential properties to ensure that their use will not jeopardize the quality of the overall application. Those properties can be listed as:

- *Statistical robustness*: the PRNG must generate bits sequences as close to real random sequences as possible. This implies that the PRNG is an implementation of a mathematical probability distribution. In other words, each member of the output family, all tuples, all sets or intervals of the same length on the distribution, are equally probable. Today, verifying these types of properties is achieved by performing statistical tests on generators. These only check some properties about random variables for predefined parameter values. Among the set of PRNGs statistical evaluation batteries, the reference one is the TestU01 [2].
- *Speed or throughput*: the PRNG must be able to reach high throughput. This is equivalent to use as less machine cycles as possible to generate one bit or one word of bits. This implies that the PRNG must perform the least operations as possible for each generation. For example, the use of multiplications has a great impact over the performance and area of hardware implementations and should be avoided [3].
- *Size*: the PRNG must be as small as possible so that it can be integrated in embedded systems (as a chip e.g.). This implies to use an inner state that is as small as possible. Also, the number and the nature of operations may be dramatic for this feature, when considering the hardware implementation on chips (FPGA,...).

In the scope of this paper, we do not consider cryptographic aspects, but we address all the other features. We propose several PRNG designs that allow to fulfill at least the first two criteria given above. Also, we keep in mind the problem of the size and we try to limit it whenever it is possible.

This article is organized as follows. The next section gives the background over PRNGs that is required to fully understand our work. Then, Section III shows how to produce PRNGs based on one jump in a N-cube, and studies them with respect to efficiency (on CPUs and on FPGAs) and to statistical properties. Section IV adds one step to this kind of combinations and shows how this additional step improves PRNGs in terms of statistical quality without reducing the efficiency. Section V provides a theoretical boundary of the mixing time of this family of generators. This demonstrates that the number of jumps can be very small without reducing the statistical quality of the PRNG. Section VI presents some numerical experiments on this family of generators. Section VII summarizes this work and provides some perspectives.

II. Background

As far as we know, it is an issue to gather statistical robustness and high speed in a same PRNG. Currently, the fastest PRNG are not robust as they do not pass the TestU01 which is a reference to evaluate statistical robustness.

On the other hand, robust PRNGs are slower as they contain either more operations or more complex ones.

Quoting [4], [5], a Random Number Generator (RNG) can be defined by a tuple $(S,\ f,\ g,\ U,\ x^0)$, in which S is the internal state space of the generator, U is the random output space, $f : S \to S$ is the transition mapping function, $g : S \to U$ is the output extractor function from a given state, and x^0 is the seed, see Fig. 1. The random output sequence is $y^1, y^2, \ldots$, where each $y^{t+1} \in U$ is generated by the two main steps described thereafter. The first step applies the transition function according to the recurrence

$$x^{t+1} = f(x^t), \tag{1}$$

where f is an algorithm. The configuration x^{t+1} is the new internal state. The second step consists in applying the generator function, further denoted as g, to the new internal state x^{t+1} leading to the output $y^{t+1} = g(x^{t+1})$.

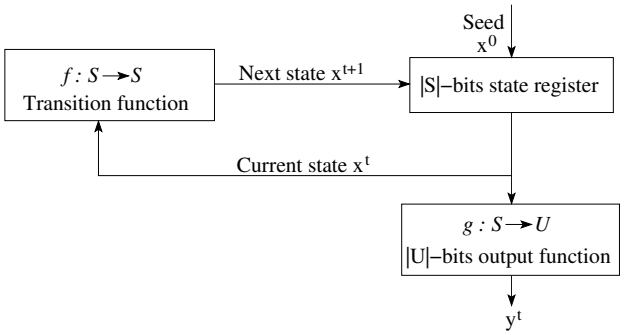

Figure 1: General Scheme of a Random Number Generator

Let $\mathbb{B} = \{0, 1\}$ be the set of Boolean values. In Listing 1, Taus88 [6] is given as an example of such Pseudo Random Number Generator. In this PRNG, the internal state is memorized as a vector $x = (x_1, x_2, x_3)$ of three 32 bits length integers in $S = \mathbb{B}^{32} \times \mathbb{B}^{32} \times \mathbb{B}^{32} \equiv \mathbb{B}^{96}$ and the output space is $U = \mathbb{B}^{32}$. The function `taus88_f()` (resp. `taus88_g()`) is the aforementioned f (resp. g) function.

It is not hard to establish in this example that

$$x_1^{t+1}[i] = f_1(x_1^t)[i] = \begin{cases} x_1^t[i-12] \text{ if } 14 \le i \le 32 \\ x_1^t[i+19] \oplus x_1^t[i+6] \text{ if } 1 \le i \le 13 \end{cases}$$

$$x_2^{t+1}[i] = f_2(x_1^t)[i] = \begin{cases} x_2^t[i-4] \text{ if } 8 \le i \le 32 \\ x_2^t[i+25] \oplus x_2^t[i+23] \text{ if } 1 \le i \le 7 \end{cases}$$

$$x_3^{t+1}[i] = f_3(x_1^t)[i] = \begin{cases} x_3^t[i-17] \text{ if } 22 \le i \le 32 \\ x_3^t[i+8] \oplus x_3^t[i+11] \text{ if } 1 \le i \le 21 \end{cases}$$

The state variable $x = (x_1, x_2, x_3)$ *jumps* from configuration $x^t \in \mathbb{B}^{96}$ to $x^{t+1} \in \mathbb{B}^{96}$ according to rules defined in previous equations. More generally, the modifications of the internal state of a PRNG, made by function f, may be seen as a way to define such jumps inside a N-cube.

The role of the g function is to pass from the internal space to the output space. Generally, this corresponds to a dimension reduction, but it may also apply a post-treatment over the internal state, which often consists in a *mixing* operation.

Listing 1: Taus88 PRNG algorithm

```
// PRNG seed (static in this case)
#define SEED 12345
// Internal state
static uint32_t x1 = SEED,
                x2 = SEED,
                x3 = SEED;
// Internal state update function
void taus88_f() {
  uint32_t b;
  // XORs and shifts
  b = (((x1 << 13) ^ x1) >> 19);
  x1 = (((x1 & 4294967294) << 12) ^ b);
  b = (((x2 << 2) ^ x2) >> 25);
  x2 = (((x2 & 4294967288) << 4) ^ b);
  b = (((x3 << 3) ^ x3) >> 11);
  x3 = (((x3 & 4294967280) << 17) ^ b);
}
// Output value computation function
uint32_t taus88_g(){
  // XOR of the three internal values
  return (x1 ^ x2 ^ x3);
}
// PRNG function to be used by programmers
uint32_t taus88(){
  taus88_f();        // Internal state update
  return taus88_g(); // PRNG Output
}
```

In the example of Taus88, the g function is such a reduce and mix function that goes from $\mathbb{B}^{96}$ to $\mathbb{B}^{32}$. In other schemes, e.g. [7], [8], the dimension reduction is combined with a *permutation* function of the bits to be produced, which improves the statistical properties. However, in the case where function g requires additional state variables, the evolution of these variables must be placed into f. Also, the memory size of these additional variables must be added to the size of S when calculating the global generator's dimension.

It therefore remains to define the elements S, f, g, and U. The work presented in this paper asserts that both f and g can be defined thanks to two *weak* PRNGs, *i.e.*, not statistically robust, but generally operating in low-dimensional spaces and extremely fast. This approach has been successfully used in the following cases [8] with $U = \mathbb{B}^{32}$, f similar to the aforementioned Taus88, and the g function composed of a XOR between the Taus88 state variables (x_1, x_2, x_3) (as in Taus88) and a simplified version of PCG32 [7] whose internal state is $S_{\text{pcg32}} = \mathbb{B}^{64}$. This PRNG has been deployed on FPGA. On this support, as far as we know, it is the generator with the highest throughput rates that pass the entire TestU01. The addressed question in this work is:

Does there exist a combination (and what kind of combination) of lightweight PRNGs whose throughput on this support is higher than the previous ones whilst passing all the statistical tests of TestU01?

III. One Jump is not Sufficient

A set of PRNGs has been selected for evaluation purpose:

- The *size* of state space S and thus the size of the output space U is the first criterion we consider. Indeed, it is in direct relation with the area that is required to be reserved on the FPGA. A priori, the smaller the size of the spaces, the smaller the area deployed on the FPGA to process data of these spaces.

- The *underlying mechanism* of pseudo-random number generation directly influences both the quality of the sequence of generated bits and the time required to generate them. It is well known that it is not efficient to implement products with large factors on FPGAs. A PRNG based on a LCG $x^{t+1} = (a.x^t + c) \mod m$ with a large a multiplier can be efficiently deployed on CPU (where the multiplication operation is not an issue anymore) but not on FPGA.
- The *ability to pass* the test TestU01. The TestU01 informs us about the intrinsic qualities of the generator that will be integrated internally into our combination approach. To fully pass the TestU01, it is not hard to understand that it is a priori better to use internally a generator that passes almost all the statistical tests of the TestU01 than to use a generator that fails most of them.

Among all PRNGs, seven have been selected according to these criteria. First of all, a set of four efficient PRNGs based on the LFSR scheme, namely LFSR113 [9], Taus88 [2], Xorshift128+ [10] and Xoroshiro128+ [11] whose internal space is lower or equal to $\mathbb{B}^{128}$. Notice that none of these generators use multipliers and all of them fail the Linear Complexity test of TestU01. For efficiency comparison purpose, we have furthermore evaluated three PRNGs that succeed the whole TestU01, namely PCG32 [7] (based on a LCG and a permutation function), Taus88+PPCG32 [8] (a combination of Taus88, for jumping in the 32-cube, and of a permutation function based on a simplified version of PCG32) and CIPRNG [12] (a combination of three Xorshift based PRNGs).

Results of this evaluation are summarized in Table I. The second column gives the size of the internal state space S. The third one specifies which part of TestU01 has failed (if any). For this family of test, each PRNG is evaluated 100 times, with a different random seed each time. Only tests that systematically fail, (*i.e.*, 100 times with p-values less than 10^{-15}) are reported here. The fourth one indicates how many cycles are required to generate 1 byte using Lemire's evaluation tool [13]. Finally, the last two columns present the throughput (in Gb/s) and the area (in Gate Equivalent) obtained with the FPGA implementation of each PRNG.

The hardware implementation of the selected PRNG are evaluated by using Xilinx Vivado tool and Digilent Zybo Z7-10 Board (PRNG of 64-bit uses Zybo Z7-20). The physical implementation results are based on two parameters, the frequency of the circuit and the output range, whose multiplication gives the throughput [12]. The gate equivalent area ((FF+LUT)×8) is only evaluated by Flip-Flop (FF) and Lookup-Table (LUT), which are the base elements for each FPGA technologies (the slice parameter is calculated differently for Xilinx or Alteras).

Because multiplications are widely used in PRNGs, they can be implemented with DSP. However, in hardware level, these arithmetic operations (especially the multiplication) are hard coded inside the tools (Xilinx) using optimized algorithms for that. Unlike the CPU or GPU, all arithmetic operations in FPGA are executed in parallel using *Distributed Arithmetic* with less space and power usage (no complex multi-tasking or multi-core used). They perform a multiply-and-add operation at the same time using most basic logic elements (LUTs in FPGA). Their size and performance depend on both the *word length* and their binary representations, regarding *dynamic range* and precision

Table I: Statistical and Efficiency Evaluation of Linear PRNGs

PRNG	Output size	Size of S	TestU01 failures (if any)	Nb cycles/bytes on CPU	Throughput (Gb/s)	Area (GE)
LFSR113	32	128	Matrix Rank Linear Comp	2.6	16.49	1984
Taus88	32	96	Matrix Rank Linear Comp	2.61	16.52	1616
PCG32	32	64	No test failed	1.86	0.29	6106
Taus88+PPCG32	32	128	No test failed	3.8	6.95	4936
CIPRNG	32	352	No test failed	4.01	8.50	10720
Xorshift128+	64	128	Linear Comp	0.9	15.77	2784
Xoroshiro128+	64	128	Matrix Rank Linear Comp	1.09	14.88	2312

It is clear from Table I that PRNGs like LFSR, Taus, Xorshift and Xoroshiro have the highest throughput and lowest areas compared to the others. These results confirm the first remark concerning the drawback of using arithmetic operations or physical accelerators such as DSP.

As an example, it can be seen that PCG32 takes 1.86 Cycle/Byte in CPU execution where physical accelerators are used to optimize 64-bit multiplications. However, in the domain of PRNGs, the goal is physical implementation for industrial uses as ASIC, in which these accelerators must be built from scratch to meet timing. Indeed, these is why in hardware implementation of PCG32, the number of cycles increases (20 cycles) with partial product, conducting to a limited throughput.

The embedded PRNG is considered here to be a black box: the internal state space, its 32-bit output s^t, are known, but there is no change in the code of this embedded generator. The studied space is $\mathbb{B}^{32}$ and the state we are interested in is a vector $x^t = (x[0]^t, \ldots, x[31]^t)$ of this space. Two jumping approaches are defined.

The first one corresponds to a jump from x^t into the complete 32-cube following the s^t strategy. Formally, we have

$$x^{t+1} = x^t \oplus s^t,$$

or, in other words, f of the equation (1) is the binary negation function restricted to items defined by s^t.

The second approach corresponds to a jump into a 32-cube in which a balanced oriented Hamiltonian cycle has been removed. It has indeed been shown that removing a Hamiltonian cycle in an N-cube does not change its strong connectivity [14]. Intuitively, the proof is based on the fact that no edge of the same Hamiltonian cycle taken in the opposite direction is removed and that this cycle allows, at least, to connect all the vertices between them. In addition, it has also been shown that the associated Markov matrix to this type of jump is also doubly stochastic [14], which is a necessary and sufficient condition for the output to be uniformly distributed. Basically, the proof is that exactly one outgoing edge is

removed from each vertex in a N-cube (whose Markov Matrix is obviously doubly stochastic) and its associated probability is reallocated to the loop over this vertex. Notice that strong connectivity is not is not guaranteed if two Hamiltonian cycles are removed (the proof is left to the reader).

Table II shows experiments on combining one PRNG with a jump, either in the complete N-cube (and denoted as *PRNG*-XOR) or in a N-cube where a Hamiltonian cycle is removed (denoted as *PRNG*-HAM). What can be first deduced is that applying a XOR does not solve any failure in Matrix Rank or Linear Comp tests. For the former, it is not a surprise. The output y^{t+1} of this kind of generator is indeed a configuration in $\mathbb{B}^{32}$ resulting of an exclusive or between the previous output y^t and the output of a combined LFSR whose recurrence equation is $x^t_{i+k} = x^t_{i+q} \oplus x^t_i$. In other words, $y^{t+1} = y^t \oplus x^t$. It is a simple exercise to prove by recurrence on t that $y^t_{i+k} = y^t_{i+q} \oplus y^t_i$ for any t *i.e.*, y is an LFSR and that such kind of generator fails the Matrix Rank test.

Table II: Combining PRNGs with a jump, in the 32-cube

PRNG	Size of S (bits)	TestU01 failures (if any)	Nb cycles/bytes on CPU	Throughput (Gb/s)	Area (GE)
LFSR113-HAM	160	Random Walk 1	5.19	9.73	3720
LFSR113-XOR		Matrix Rank, Linear Comp	4.00	16.09	2240
Taus88-HAM	128	Random Walk 1	4.81	9.66	3352
Taus88-XOR		Matrix Rank Linear Comp	3.61	16.15	1927

In terms of throughput and area, the use of Hamiltonian cycles induces a decrease of throughput, due to an important increase of area. Indeed, the impact over the throughput is lighter than over the area. The important increase in area is due to the hardware implementation of the cycles functions. Its limited impact over the throughput comes from using only 8-bits Boolean arithmetic operations in parallel without any hard macro as memories or DSPs.

IV. One jump and one mixing based PRNG

Section IV-A starts with presenting the proposed combining of PRNGs. Section IV-B shows how the allowed bits in the PRNGs were technically stored. Finally, Section IV-C presents candidates for combinations.

A. General Scheme Proposal

According to the previous results, we have designed a general scheme of PRNG that includes the removing of a Hamiltonian cycle in the N-cube of internal state. This scheme involves two (weak) PRNGs that respectively correspond to f and g functions in Fig. 1. It can be described as in Listing 2.

Concerning the building of the strategy, *i.e.*, the `st` value that is XORed to the current internal state, there are two possibilities that depend on the function `allowedBits`. In the classical case, where there is no constraint and the jump can be everywhere in the N-cube, the `allowedBits` function always returns a 32-bits word with all bits with value 1. In the other case, where a Hamiltonian cycle is removed, the `allowedBits` function must return a 32-bits word with ones everywhere save at some positions corresponding to the forbidden dimensions for the current move. As these forbidden dimensions depend on the current state, *i.e.* the current position in the N-cube, the `allowedBits` function must take the current state as parameter. This additional constraint of forbidden dimensions represents an increase in complexity that can be useful for the robustness of the final PRNG.

Listing 2: General PRNG scheme using two inner PRNGs

```
// Get the current moving strategy in the n-cube
st = state.str(); // str() is a 32-bits PRNG
// Get the mask of allowed bits
// from the Hamiltonian cycle
ab = state.allowedBits(state.currentValue);
// Updating of the strategy according
// to the allowed bits
st = st & ab;
// New internal state deduced from a XOR between
// the current state and the moving strategy
state.currentValue = (state.currentValue ^ st);
// PRNG output deduced from the current state
// sent through the mixing function
return state.mix(state.currentValue);
```

B. Details over the `allowedBits` function

As described above, the `allowedBits` function must return the mask of dimensions of the N-cube along which a move is possible according to the current position.

To build this mask, a Hamiltonian cycle would be required in the 32-cube. However, although it would be possible to find such a cycle by using one of the generation algorithms designed in [15], [16], this is not done in practice, due to the very large state space. Indeed, the description of a Hamiltonian cycle requires to store at least the dimension index (*i.e.*, 2^{32} values for a cycle in a 32-cube) for each vertex in the N-cube, which is not pertinent in the current context.

So, in order to get smaller storage requirements, the 32-bits word is divided into sub-words of k-bits each (k=8 or 16). For 16-bits sub-words, two Hamiltonian cycles in 16-cube are needed, requiring 2^{16} indices each, leading to a total storage of 2^{17} values. For 8-bits sub-words, four Hamiltonian cycles are needed in 8-cube, leading to only 2^{10} values to store. Although N-cubes of any dimension between 2 and 32 may be used, it is better, for statistical quality, to use sufficiently large dimensions as well as dimensions that are powers of 2. Indeed, such dimensions of N-cubes are the only ones to provide totally balanced Hamiltonian cycles. This is why we consider in this work only N-cubes of dimension 8 or 16.

The direct consequence over the `allowedBits` function is that it must manage several cycles (two or four) instead of only one. This is done by concatenating the k-bits cycles states in order to obtain the 32-bits internal state of the PRNG, as depicted in Figure 2.

In classical programming language such as C/C++, such word concatenation can be directly achieved by using the `union` structure, as shown in Listing 3. This allows us to get access to different parts of the same memory location, which is the PRNG internal state in our case.

Finally, according to performance aspect, it is more efficient to directly store the mask of allowed bits per N-cube vertex

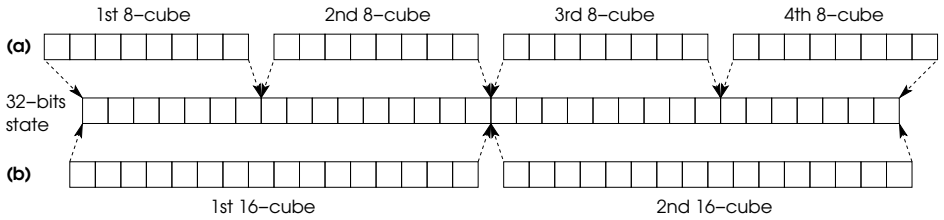

Figure 2: Concatenation of several k-bits cycles states to compose the 32-bits PRNG internal state: (a) Four 8-bits cycles (b) Two 16-bits cycles.

(denoted as `ab` in Listing 2), as it avoids building that mask dynamically during the PRNG process.

Listing 3: Union structure used as a polymorphism of the PRNG internal state.

```
typedef union {
  // 32–bits integer representing
  // the PRNG internal state
  uint32_t i;
  // two 16–bits integers representing
  // the cycles in 16–cubes
  uint16_t s[2];
  // four 8–bits integers representing
  // the cycles in 8–cubes
  uint8_t  b[4];
} Union;
```

Following this construction, the `allowedBits` function can be described by Listing 4. In this code sample, the cycles are stored in two dimensional arrays whose first dimension is the cycle number, and the second dimension is the vertex number in the N-cube. The content of array cell `[i][j]` is the mask of allowed bits at vertex `j` in cycle `i`.

Listing 4: `allowedBits` function with the current state parameter `curState`

```
// Union used to manage the cycles
Union u;
// Setting the current state into the union
u.i = curState; // 32–bits integer
// Getting the allowed bits for the
// respective states of each cycle
// (example with two 16–bits cycles)
u.s[0] = cycle[0][u.s[0]]; // Allowed bits for vertex
                           // u.s[0] in the 1st cycle
u.s[1] = cycle[1][u.s[1]]; // Allowed bits for vertex
                           // u.s[1] in the 2nd cycle
// Concatenation of allowed bits of the cycles
return u.i;
```

The obtained PRNG scheme allows us to easily build many PRNGs by combining two 32-bits PRNGs and a given number of Hamiltonian cycles in k-cubes. However, in this scheme, the use of multiple cycles could be seen as a weakness, due to their limited size (implying small periods) and their independent use. Nevertheless, this can be overcome by mixing their respective evolutions. This is done by circularly shifting the cycles at each execution of the `allowedBits` function, as depicted in Listing 5. In this algorithm, only one additional variable is required to store the current shifting position, as well as a few operations on the cycles indices.

From a probabilistic point of view, we will show in the next section that when one Balanced Hamiltonian cycle per cube is removed, it is sufficient to invert all the bits on average 4 iterations, *i.e.*, to reach any vertex. The same is obviously true when, in addition, at each iteration, the balanced Hamiltonian cycle of a another 1-byte group is used.

Table III: Combinations of PRNGs used in the general scheme

Strategy	Mixing
LFSR	Taus88
Taus	LFSR
Xorshift128+	Xorshift128+
Xoroshiro128+	Xoroshiro128+

Listing 5: `allowedBits` function with cycles shifts

```
// Current shift of the cycles
static char cycleShift = 0;
...
// Shift update for two cycles (0 and 1)
cycleShift = 1 – cycleShift;
...
// Getting the allowed bits for
// the respective states of each cycle
// (example with two 16–bits cycles)
u.s[0] = cycle[cycleShift][u.s[0]]; // Allowed bits for
                  // vertex u.s[0] in cycle "cycleShift"
u.s[1] = cycle[1 – cycleShift][u.s[1]]; // Allowed bits
                 // for vertex u.s[1] in the other cycle
...
```

In the following subsection, we present different PRNG constructions, based on this general scheme, and for each one we consider two versions, with and without cycles shifts.

C. Considered combinations of PRNGs

As described previously, there are many possible variants to build a PRNG by using our general scheme. In addition to the choices of strategy and mixing functions, comes the number of sub-words (N-cubes) to decompose the PRNG output word. In the scope of this study, we consider only N-cubes of dimension 8 or 16. Also, as our general scheme requires two 32-bits words internally, one for the strategy and one for the mixing, we have considered to use one 64-bits PRNG instead of two 32-bits PRNGs. For performance sake, we have selected some very fast 64-bits PRNGs, given in Table I.

Selected combinations given in Table III consider both variants with and without cycle shifts.

Also, in order to preserve performance, 64-bits PRNGs are always used for both parts of the global PRNG (strategy and mixing), by splitting its output in two parts, as already explained. Moreover, only the PRNGs with reduced storage costs are kept.

Finally, for each combination in Table III, we consider three variants according to the presence and size of Hamiltonian cycles: 2×16-bits, 4×8-bits or none.

V. Convergence Rate

The are two main approaches to measure the convergence rate of a Markov Chain, *i.e.*, how fast it converges to the stationary distribution: the mixing time and strong stationary times. The mixing time of a Boolean function in the N-cube provides accurate information on the average number of iterations that are sufficient to provide an output with a uniform distribution, to the nearest epsilon. The use of Markov chains makes it possible to demonstrate that this average number of iterations is constant. In a previous work, in which walking in a N-cube (each move is along only one dimension at a time) was considered instead of jumping (what we do in this article), it has been proven that a upper bound of this number was in $\mathsf{N}\times\log(\mathsf{N})$, where N is the number of produced bits. Strong stationary times mathematically ensure that the limit distribution is reached.

Using a classical result on symmetric and ergodic Markov chains, walking or jumping in a N-cube where a Hamiltonian cycle has been deleted induces a Markov chain M whose stationary distribution π is the uniform one (see [14] for details). Let us then recall some probabilistic definitions of mixing and strong stationary times.

First of all, let be given two distributions μ and π on the same set Ω, the Total Variation distance $\|\mu-\pi\|_{\mathrm{TV}}$ is defined by:

$$\|\mu-\pi\|_{\mathrm{TV}}=\max_{A\subset\Omega}|\mu(A)-\pi(A)|.$$

Let then $M(x,\cdot)$ be the distribution induced by the x-th row of the Markov matrix M. If the Markov chain induced by M has a stationary distribution π, then we define

$$d(t)=\max_{x\in\Omega}\|M^t(x,\cdot)-\pi\|_{\mathrm{TV}}.$$

Finally, let ε be a positive number, the *mixing time* with respect to ε is given by

$$t_{\mathrm{mix}}(\varepsilon)=\min\{t\mid d(t)\leq\varepsilon\}.$$

Intuitively, it defines the smallest iteration number that is sufficient to provide a deviation lesser than ε to the stationary distribution, which is here the uniform one. It is known that $t_{\mathrm{mix}}(\varepsilon)\leq t_{\mathrm{mix}}(1/4)\log_2(\varepsilon^{-1})$.

Intuitively, strong stationary time in a Markov chain is the instant when the stationary distribution is reached. Let τ_{stop} be the first time all the elements of $[\![1,\mathsf{N}]\!]$ could have been inverted. More precisely, for any element x in the N-cube (where an Hamiltonian cycle has been removed), we denote by $\mathrm{mv}(x)$ the set of elements j of $\{1,\ldots,N\}$ such that (x,y_j) (where y_i is obtained from x by just switching the i-th bit) is an edge in this modified N-cube. For a Markov Chain (x_i) in the modified N-cube, the random variable τ_{stop} is defined as the minimum t such that $\cup_{i=0}^{t}\mathrm{mv}(x_i)=\{1,\ldots,N\}$. Each bit of $x_{\tau_{\mathrm{stop}}}$ is uniformly distributed, proving that τ_{stop} is a strong stationary time. We have the following theorem.

Theorem 1. *$E[\tau_{\mathrm{stop}}]$ is less than 4.*

Proof. Let us consider a N-cube where a balanced Hamiltonian cycle has been removed. We re-use some notations introduced in [15] to define a balanced Hamiltonian cycle. Let $L=w_1,w_2,\ldots,w_{2^{\mathsf{N}}}$ be the sequence of a N-bits cyclic Gray code. The transition sequence $S=s_1,s_2,\ldots,s_{2^{\mathsf{N}}}$, $1\leq i\leq 2^{\mathsf{N}}$, indicates which bit position changes between code-words at index i and $i+1$ modulo 2^{N}. The *transition count* function $TC_{\mathsf{N}}:\{1,\ldots,\mathsf{N}\}\to\{0,\ldots,2^{\mathsf{N}}\}$ gives the number of times i occurs in S, *i.e.*, the number of times the bit i has been switched in L.

Let N in $\mathbb{N}^*$, and a_{N} be defined by $a_{\mathsf{N}}=2\left\lfloor\frac{2^{\mathsf{N}}}{2\mathsf{N}}\right\rfloor$, where $\lfloor x\rfloor$ denotes the greatest integer less than or equal to the real number x. A cyclic Gray code is balanced if and only if for any bit position i, $1\leq i\leq\mathsf{N}$,

$$a_{\mathsf{N}}\leq TC_{\mathsf{N}}(i)\leq a_{\mathsf{N}}+2.$$

In the first stage, $\mathsf{N}-1$ bit can be modified. Since jumps are executed in a $\mathsf{N}-cube$ where a Hamiltonian path has been removed and without loss of generality, we can consider that the first bit cannot be switched. Let us consider the jump done and let C be the reached configuration. It remains thus to calculate how many jumps are required to have the possibility to modify the first bit.

In the configuration C, the probability that the first bit cannot be switched again is p' where

$$\frac{a}{2^{\mathsf{N}}}\leq p'\leq\frac{a+2}{2^{\mathsf{N}}}.$$

The probability of the complementary event, *i.e.*, to be able to switch the first bit, is $p=1-p'$, which can be bounded as follows.

$$1-\frac{a+2}{2^{\mathsf{N}}}\leq p\leq 1-\frac{a}{2^{\mathsf{N}}}.$$

The random variable that counts the number of jumps required to switch this first bit follows a geometric distribution of success p. Its expected value is $E=\frac{1}{p}$ which can be bounded as follows:

$$\begin{aligned}&\frac{1}{1-\frac{a}{2^{\mathsf{N}}}}\leq E\leq\frac{1}{1-\frac{a+2}{2^{\mathsf{N}}}},\\ \textit{i.e.,}\quad&\frac{2^{\mathsf{N}}}{2^{\mathsf{N}}-a}\leq E\leq\frac{2^{\mathsf{N}}}{2^{\mathsf{N}}-a-2}.\end{aligned}$$

Since $a_{\mathsf{N}}=2\left\lfloor\frac{2^{\mathsf{N}}}{2\mathsf{N}}\right\rfloor$, a_{N} is lower than $\frac{2^{\mathsf{N}}}{\mathsf{N}}$, and thus,

$$2^{\mathsf{N}}-a_{\mathsf{N}}-2\geq 2^{\mathsf{N}}\left(1-\frac{1}{\mathsf{N}}-\frac{2}{2^{\mathsf{N}}}\right),$$

leading to,

$$E\leq\frac{2^{\mathsf{N}}}{2^{\mathsf{N}}-a_{\mathsf{N}}-2}\leq\frac{1}{1-\frac{1}{\mathsf{N}}-\frac{2}{2^{\mathsf{N}}}}\leq\frac{\mathsf{N}\times 2^{\mathsf{N}}}{\mathsf{N}\times 2^{\mathsf{N}}-2^{\mathsf{N}}-2N},$$

if N is not equal to 2.

It is not hard to verify that over $]2,\infty)$, the function $x\mapsto\dfrac{x\times 2^x}{x\times 2^x-2^x-2x}$ is decreasing . Thus E is less than $\dfrac{3\times 2^3}{3\times 2^3-2^3-2\times 3}=2.4$. In the specific case where $\mathsf{N}=2$, the Gray code is totally balanced, *i.e.*, $TC_2=2$. Hence $p'=p=\frac{1}{2}$ and E is 2.

All cases lead to the conclusion that E is less than 3 and therefore that $E[\tau_{\text{stop}}]$ is less than 4, which ends the proof. □

By using now t we have the Markov inequality:

$$P(\tau > t) \leq \frac{E(\tau)}{t} \leq \frac{4}{t}.$$

Therefore, using this inequality for $t = 16$ and [17, Lemma 6.13], it follows that $t_{\text{mix}}(\frac{1}{4}) \leq 16 = O(1)$, $t_{\text{mix}}(\frac{1}{4})$ is bounded by a constant independent of N. The inequality also points out that the probability of not achieving the equilibrium decreases at least as $\frac{1}{t}$ with the number t of steps, showing a fast convergence.

VI. Experiments

For the experimental evaluation, we have used the TestU01 C library [2]. Our PRNG scheme has been fully implemented with the possibility to choose the strategy and mixing functions, among the four standard PRNGs given in Table III. Also, a file of Hamiltonian cycles can be specified to activate constrained jumps in the N-cube. Indeed, as we need the allowed bits for any vertex in the N-cube, the file can directly contain the cycles expressed under that form. However, for hardware implementation on FPGA, it is more efficient to express the cycles as Boolean functions, as the use and accesses to static arrays is much more area and time consuming.

Results are summarized in Table IV. The names of failed test are listed in column TestU01 when there are just a few, otherwise only the number of failed tests is given. Otherwise, the column contains *OK*. In the 6^{th} column reports the average number of cycles to generate 1 byte on an Intel E5-2640@2Ghz CPU, based on 1000 generations of 10^6 bytes. The areas obtained after a place and route stage are reported in the last column.

As can be seen, the combination with LFSR as the strategy and Taus as the mixing function (L-*-T) is not robust, whatever the cycles are used or not. However, the inverse combination (T-*-L) is much better. This shows us that the choice for the strategy and the mixing is not symmetric. Also, the second combination shows that the use of Hamiltonian cycles can bring robustness to a PRNG. Finally, the other combinations pass all the tests but their respective internal states are larger (128 bits) than those of T-C-L (96 bits).

Concerning the performance aspect in terms of cycles per generated byte, the use of Hamiltonian cycles implies overheads that depend on the dimension of the underlying N-cube. However, those overheads stay quite limited for 8-bits cycles, especially when combined with the fastest PRNGs like Xorshift128+ and Xoroshiro128+ (respectively 13% and 14%).

Concerning hardware implementations, the different hardware combinations can be classified in two categories based on the results in Table IV:

a) Combining PRNGs with a jump in a complete N*-cube:* The hardware implementations are based on strategy and mixing words generations in parallel, followed by the updating of the internal state and the generation of the output word. In such architecture, the critical path is only related to either the strategy or the mixing function. Indeed, Taus88-XOR-LFSR has the highest throughput compared to the other combinations, but it fails the TestU01. Both combinations Xorshift128-XOR-Xorshift128 and Xoroshiro128-XOR-Xoroshiro128 pass the test but with lower throughput. However, they provide better performance than some single PRNGs given in Table I and all the combinations listed in Table II.

b) Combining PRNGs with a jump in a N*-cube where a Hamiltonian cycle is removed:* As mentioned before, these combined PRNGs apply the jump in a set of N-cubes with or without cycles shifts. First, the hardware design of the combined PRNGs is based on three steps: generating the strategies for each N-cube (only one inner PRNG call), splitting the internal state in 4 or 2 blocks depending on the size of H-Cycles (8 or 16 bits), and updating the internal state by applying the jump only over the allowed bits given by the H-Cycles. When based on 8-cubes (resp. on 16-cubes), cycles are expressed exclusively with Boolean arithmetic operations. Therefore, implementing two 16-bits H-Cycles in FPGA requires 2 SRAM dual-port memories. Also, applying shifts to the H-Cycles composing the internal state requires more logic and implies an additional delay to the PRNG output.

Finally, when looking at the combinations that pass TesTU01, Taus88-HAM8-LFSR without cycles shifts (T-H8-L) provides the best throughput, with 9.32Gb/s. It is significantly better than the CIPRNG (8.5Gb/s) given in Table II although T-H8-L has a smallest internal size (128 bits instead of 352 bits) and a smaller area (5336 GE instead of 10720). By the way, the area of T-H8-L is one of the smallest among all the PRNGs that pass TestU01.

VII. Conclusion

When performing numerical simulations, used pseudo random numbers must be flawless in order to avoid any bias in the results of the simulation. Also, they must be quickly produced so that the simulation can be carried out as fast as possible.

This work has first shown that any PRNG can be seen as a form of jumping into an N-cube whose dimension defines the internal space. The way of jumping is specific to each PRNG, the important aspect being that it is as "unpredictable" as possible while remaining sufficiently fast.

Our approach is twofold. The former is a jump into a N-cube in which a balanced Hamiltonian path has been removed guided by an effective generator (Taus88 for example). The latter one produces the output by mixing the current state with another generator (LFSR113 for example). The result is a generator that successfully passes the whole TestU01, which is considered to be the most difficult for PRNGs.

If this combination of jumping and mixing had already been sketched in a previous work, we have greatly strengthened it in this work. From a theoretical point of view, we demonstrated that the mixing time was constant (while it was in $\mathsf{N}\log(\mathsf{N})$) and that we could get rid of more complex generators (PCG32 for example), while continuing to pass the TestU01. We have also conducted an exhaustive efficiency study of this family of generators on both CPU and FPGA. On the first platform,

Table IV: Results for each combination and variant in the general scheme

Strategy	H-Cycles	Cycles shifts	Mixing	TestU01	Nb cycles/byte on CPU	FPGA Throughput (Gb/s)	Area (GE)
LFSR	none	N/A	Taus	MatrixRank LinearComp	5.20	15.78	3912
LFSR	2×16-bits	YES	Taus	MatrixRank	8.01	4.54	5520
LFSR	2×16-bits	NO	Taus	MatrixRank	7.58	4.92	5504
LFSR	4×8-bits	YES	Taus	14 tests	7.16	6.34	7640
LFSR	4×8-bits	NO	Taus	14 tests	6.59	8.99	5336
Taus	none	N/A	LFSR	MatrixRank LinearComp	5.10	15.35	3848
Taus	2×16-bits	YES	LFSR	OK	8.24	4.51	5368
Taus	2×16-bits	NO	LFSR	OK	7.76	4.90	5104
Taus	4×8-bits	YES	LFSR	OK	7.15	6.01	7648
Taus	4×8-bits	NO	LFSR	OK	6.43	**9.32**	5336
Xorshift128+	none	N/A	Xorshift128+	OK	4.63	7.63	4048
Xorshift128+	2×16-bits	YES	Xorshift128+	OK	7.26	4.40	6144
Xorshift128+	2×16-bits	NO	Xorshift128+	OK	7.03	4.35	5880
Xorshift128+	4×8-bits	YES	Xorshift128+	OK	5.70	5.66	7888
Xorshift128+	4×8-bits	NO	Xorshift128+	OK	5.25	7.27	5552
Xoroshiro128+	none	N/A	Xoroshiro128+	OK	4.40	7.37	3088
Xoroshiro128+	2×16-bits	YES	Xoroshiro128+	OK	7.22	4.51	5384
Xoroshiro128+	2×16-bits	NO	Xoroshiro128+	OK	6.98	4.75	5112
Xoroshiro128+	4×8-bits	YES	Xoroshiro128+	OK	5.37	6.15	7232
Xoroshiro128+	4×8-bits	NO	Xoroshiro128+	OK	5.02	7.21	4784

we were able to provide a whole family of PRNGs that pass the complete TestU01, but which is twice as slow as PCG32. On FPGA, to the best of our knowledge, we obtain the fastest generator in the world (9.32Gb/s) that passes the entire TestU01. This result significantly improves our previous score which was 8.5Gb/s. This is very promising and let us think that it is possible to achieve even smaller PRNGs that pass TestU01 with throughput beyond 10Gb/s.

Future works cover many aspects. A first track consists in generating more bits (128 bits e.g.). Then, we plan to further study XORShift-based generators to exhibit minimum conditions to pass the TestU01, while improving the flow rate. Also, exploring the reduction of the internal state size as well as the number of operations should lead to faster generators.

ACKNOWLEDGMENT

This work is funded by the Labex ACTION program (contract ANR-11-LABX-01-01). Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté.

REFERENCES

[1] Rany El Haddad, Christian Lécot, Pierre L'Ecuyer, and N. Nassif. Quasi-monte carlo methods for markov chains with continuous multi-dimensional state space. *Mathematics and Computers in Simulation*, 81(3):560–567, 2010.

[2] Pierre L'Ecuyer and Richard Simard. Testu01: A c library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33(4):22:1–22:40, August 2007.

[3] M Anwar Hasan and Christophe Negre. Sequential multiplier with sub-linear gate complexity. *Journal of Cryptographic Engineering*, 2(2):91–97, 2012.

[4] Pierre L'Ecuyer. Random numbers for simulation. *Communications of the ACM*, 33(10):85–97, 1990.

[5] Mohammed Bakiri, Christophe Guyeux, Jean-François Couchot, and Abdelkrim Kamel Oudjida. Survey on hardware implementation of random number generators on FPGA: theory and experimental analyses. *Computer Science Review*, 27:135–153, 2018.

[6] Pierre L'Ecuyer. Maximally equidistributed combined tausworthe generators. *Mathematics of Computation of the American Mathematical Society*, 65(213):203–213, 1996.

[7] Melissa E O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Transactions on Mathematical Software*, 2014.

[8] Mohammed Bakiri, Christophe Guyeux, Jean-François Couchot, Luigi Marangio, and Stefano Galatolo. A hardware and secure pseudorandom generator for constrained devices. *IEEE Trans. Industrial Informatics*, 14(8):3754–3765, 2018.

[9] Pierre L'Ecuyer. Tables of maximally equidistributed combined lfsr generators. *Mathematics of Computation of the American Mathematical Society*, 68(225):261–269, 1999.

[10] Mutsuo Saito and Makoto Matsumoto. Xorshift-add: a variant of xorshift. http://www.math.sci.hiroshima-u.ac.jp/∼m-mat/MT/XSADD, 2014.

[11] David Blackman and Sebastiano Vigna. Scrambled linear pseudorandom number generators. *CoRR*, abs/1805.01407, 2018.

[12] Mohammed Bakiri, Jean-François Couchot, and Christophe Guyeux. CIPRNG: A VLSI family of chaotic iterations post-processings for $\mathbb{F}_2$-linear pseudorandom number generation based on zynq mpsoc. *IEEE Trans. on Circuits and Systems*, 65-I(5):1628–1641, 2018.

[13] Daniel Lemire and Melissa E O'Neill. Xorshift1024*, xorshift1024+, xorshift128+ and xoroshiro128+ fail statistical tests for linearity. *Journal of Computational and Applied Mathematics*, 350:139–142, 2019.

[14] Sylvain Contassot-Vivier, Jean-François Couchot, Christophe Guyeux, and Pierre-Cyrille Héam. Random walk in a n-cube without hamiltonian cycle to chaotic pseudorandom number generation: Theoretical and practical considerations. *I. J. Bifurcation and Chaos*, 27(1):1–18, 2017.

[15] Sylvain Contassot-Vivier and Jean-François Couchot. Canonical Form of Gray Codes in N-cubes. In Alberto Dennunzio, Enrico Formenti, Luca Manzoni, and Antonio E. Porreca, editors, *23th International Workshop on Cellular Automata and Discrete Complex Systems (AUTOMATA)*, volume LNCS-10248 of *Cellular Automata and Discrete Complex Systems*, pages 68–80, Milan, Italy, Jun 2017. Springer International Publishing. Part 2: Regular Papers.

[16] Sylvain Contassot-Vivier, Jean-François Couchot, and Pierre-Cyrille Héam. Gray codes generation algorithm and theoretical evaluation of random walks in n-cubes. *Mathematics*, 6(6), 2018.

[17] Yuval Peres D.A. Levin and Elizabeth L. Wilmer. *Markov Chain and Mixing Times*. American Mathematical Society, 2008.