Further scramblings of Marsaglia's `xorshift` generators

Sebastiano Vigna, Università degli Studi di Milano, Italy

`xorshift*` generators are a variant of Marsaglia's `xorshift` generators that eliminate linear artifacts typical of generators based on $\mathbf{Z}/2\mathbf{Z}$-linear operations using multiplication by a suitable constant. Shortly after high-dimensional `xorshift*` generators were introduced, Saito and Matsumoto suggested a different way to eliminate linear artifacts based on addition in $\mathbf{Z}/2^{32}\mathbf{Z}$, leading to the `XSadd` generator. Starting from the observation that the lower bits of `XSadd` are very weak, as its reverse fails systematically several statistical tests, we explore `xorshift+`, a variant of `XSadd` using 64-bit operations, which leads, in small dimension, to extremely fast high-quality generators.

Categories and Subject Descriptors: G.3 [**PROBABILITY AND STATISTICS**]: Random number generation; G.3 [**PROBABILITY AND STATISTICS**]: Experimental design

General Terms: Algorithms, Experimentation, Measurement

Additional Key Words and Phrases: Pseudorandom number generators

1. INTRODUCTION

`xorshift` generators are a simple class of pseudorandom number generators introduced by Marsaglia [2003]. While it is known that such generators have some deficiencies [Panneton and L'Ecuyer 2005], the author has shown recently that high-dimensional `xorshift*` generators, which scramble the output of a `xorshift` using multiplication by a constant, pass the strongest statistical tests of the TestU01 suite [L'Ecuyer and Simard 2007]. Such generators use just eight logical operations, one addition and one multiplication by a constant, and, as a result, they are some of the fastest known high-quality generators.[1]

Shortly after the introduction of high-dimensional `xorshift*` generators, Saito and Matsumoto [2014] proposed a different way eliminate linear artifacts: instead of multiplying the output of the underlying `xorshift` generator (based on 32-bit shifts) by a constant, they add it (in $\mathbf{Z}/2^{32}\mathbf{Z}$) with the previous output. Since the sum in $\mathbf{Z}/2^{32}\mathbf{Z}$ is not linear over $\mathbf{Z}/2\mathbf{Z}$, the result should be free of linear artifacts.

Their generator `XSadd` has a state space of 128 bits. However, while `XSadd` passes the BigCrush test, its *reverse* fails systematically the LinearComp, MatrixRank, MaxOft and Permutation test of BigCrush, which highlights a significant weakness in its lower bits.

In this paper, leveraging the theoretical and experimental data about `xorshift` generators contained in [Vigna 2014], we study `xorshift+`, a family of generators based on the idea of `XSadd`, but using 64-bit operations. In particular, we propose a tightly coded `xorshift128+` generator that does not fail systematically any test from the BigCrush suite of TestU01 (even reversed) and generates 64 pseudorandom bits in 1.10 ns on an Intel® Core™ i7-4770 CPU @3.40GHz (Haswell). It is the fastest generator we are aware of with such empirical statistical properties.

The software used to perform the experiments described in this paper is distributed by the author under the GNU General Public License. Moreover, all files generated during the experiments are available from the author.

[1] A constantly updated comparison of recent pseudorandom number generators can be found at `http://prng.di.unimi.it/`.

This work is supported the EU-FET grant NADINE (GA 288956).
Author's addresses: Sebastiano Vigna, Dipartimento di Informatica, Università degli Studi di Milano, via Comelico 39, 20135 Milano MI, Italy.

2. AN INTRODUCTION TO xorshift GENERATORS

The basic idea of `xorshift` generators is that the state space is modified by applying repeatedly a shift and an exclusive-or (xor) operation. In this paper we consider 64-bit shifts and state spaces of 2^n bits, with $n \geq 6$. We usually append n to the name of a family of generators when we need to restrict the discussion to a specific state-space size.

In linear-algebra terms, if L is the 64×64 matrix on $\mathbf{Z}/2\mathbf{Z}$ that effects a left shift of one position on a binary vector (i.e., L is all zeroes except for ones on the principal subdiagonal) and if R is the right-shift matrix (the transpose of L), each left/right shift/xor can be described as a linear multiplication by $\bigl(1 + L^s\bigr)$ or $\bigl(1 + R^s\bigr)$, respectively, where s is the amount of shifting.[2]

As suggested by Marsaglia [2003], we use always three low-dimensional 64-bit shifts, but locating them in the context of a larger matrix of the form[3]

$$M = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & (1+L^a)(1+R^b) \\ 1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \cdots & 1 & (1+R^c) \end{pmatrix}.$$

It is useful to associate with a linear transformation M its *characteristic polynomial*

$$P(x) = \det(M - x).$$

The associated generator has maximum-length period if and only if $P(x)$ is primitive over $\mathbf{Z}/2\mathbf{Z}$. This happens if $P(x)$ is irreducible and if z has maximum period in the ring of polynomial over $\mathbf{Z}/2\mathbf{Z}$ modulo $P(x)$, that is, if the powers z, z^2, $\ldots$, z^{2^n-1} are distinct modulo $P(x)$. Finally, to check this condition is sufficient to check that

$$x^{(2^n-1)/p} \neq 1 \mod P(x)$$

for every prime p dividing 2^{n-1} [Lidl and Niederreiter 1994].

The *weight* of $P(x)$ is the number of terms in $P(x)$, that is, the number of nonzero coefficients. It is considered a good property for generators of this kind that the weight is close to $n/2$, that is, that the polynomial is neither too sparse nor too dense [Compagner 1991].

3. xorshift+ GENERATORS

It is known that `xorshift` generators exhibit a number of linear artifacts, which results in failures in TestU01 tests like MatrixRank, LinearComp and HammingIndep. Nonetheless, very little is necessary to eliminate such artifacts: Marsaglia [2003] suggested multiplication by a constant, which is the approach used by `xorshift*` [Vigna 2014], or combination with an additive *Weyl generator*, which is the approach used by Brent [2007] in his `xorgens` generator.

The approach of `XSadd` can be thought of a further simplification of the Weyl generator idea: instead of keeping track of a separate generator, `XSadd` adds (in $\mathbf{Z}/2^{32}\mathbf{Z}$) consecutive outputs of an underlying `xorshift` generator. In this way, we introduce a nonlinear operation without enlarging the state space. In practice, this amounts to returning the sum of the currently updated word and of the lastly updated word of the state space.

[2]A more detailed study of the linear algebra behind `xorshift` generators can be found in [Marsaglia 2003; Panneton and L'Ecuyer 2005].

[3]We remark that `XSadd` uses a slightly different matrix, in which the bottom right element is $1 + L^c$.

Saito and Matsumoto [2014] claim that `XSadd` does not fail systematically any BigCrush test. This is true of the generator, but not of its *reverse* (i.e., the generator obtained by reversing the bits of the output). Testing the reverse is important because of the bias towards high bits of TestU01: indeed, the reverse of `XSadd` fails systematically a number of tests, including some that are not due to linear artifacts, suggesting that it its lower bits are very weak.

We are thus going to study the `xorshift+` family of generators, which is built on the same idea of `XSadd` (returning the sum of consecutive outputs of an underlying `xorshift` generator) but uses 64-bit shifts and the high-dimensional transition matrix proposed by Marsaglia. In this way we can leverage the knowledge gathered about high-dimensional `xorshift` generators developed in [Vigna 2014].

4. FULL-PERIOD GENERATORS

First of all, we must prove that `xorshift+` generators have full period if the underlying `xorshift` generator has. This is not obvious, as combining bits of a generator might in principle generate a shorter period.[4] We know that every bit of the state space of a full-period `xorshift` generator has full period, as it satisfies a full-period linear recurrence [Brent 2004; Niederreiter 1992], but this is not sufficient *per se* to guarantee full period of a combination of the bits of the state space: for instance, the 2-bit generator with state transformation

$$00 \to 10 \to 11 \to 01 \to$$

has a period of length 4 both in the upper and in the lower bit. Nonetheless, if we add (or xor) the bits we obtain the sequence

$$0 \to 1 \to 0 \to 1 \to,$$

which has a shorter period.

Note, however, that the generator obtained by xoring the last two outputs of a full-period `xorshift` generator gives actually the same periodic sequence of 64-bit values (the sequence is the one generated using $\boldsymbol{x}+\boldsymbol{x}T$ as seed, where T is the transition matrix). We just remark that the lowest bit output by a `xorshift+` generator is exactly the lowest bit that would be output replacing the sum with a xor, and conclude that a `xorshift+` generator based on an underlying full-period `xorshift` generator has full period.

5. CHOOSING THE SHIFTS

We already know from Vigna [2014] a choice of shifts for full-period generators with 1024 or 4096 bits of space. In this paper, however, we want to explore the idea of `xorshift+` generators with 128 bits of space to provide an alternative to `XSadd` that is free of its statistical flaws, and faster on modern 64-bit CPUs. While generators with a larger state space are essentially in large-scale parallel computations, finding generators with a small state space, strong statistical properties and speed comparable with that of a linear congruential generator (still too often commonly used in practice) is an interesting research goal.

We thus computed shifts yielding full-period generators; in particular, we computed all full-period shift triples such that a is coprime with b and $a + b \leq 64$ (there are 272 such triples). We looked for full period both using characteristic polynomials and using matrix exponentiation, so to double-check our results.

We then ran experiments following the protocol used in [Vigna 2014], which we briefly recall. We *sample* generators by executing a battery of tests from TestU01, a framework for testing pseudorandom number generators developed by L'Ecuyer and Simard [2007]. We start at 100 different seeds that are equispaced in the state space. For instance, for a 64-bit

[4] No such proof, to the best of our knowledge, has been published for `XSadd`.

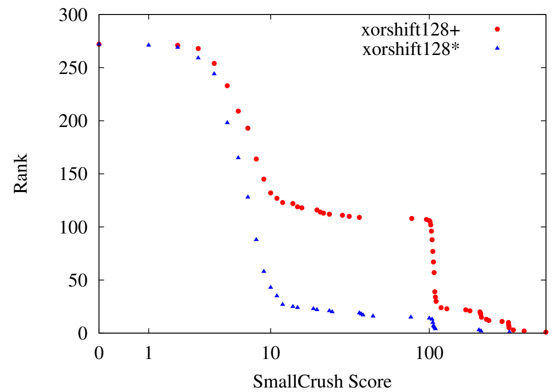

Fig. 1. Score-rank plot of the distribution of combined SmallCrush scores for the 272 full-period `xorshift128+` and `xorshift128*` generators.

state space we use the seeds $1 + i\lfloor 2^{64}/100 \rfloor$, $0 \le i < 100$. The tests produce a number of statistics, and we use the number of failed tests as a measure of low quality.

We consider a test failed if its p-value is outside of the interval $[0.001\,..\,0.999]$. This is the interval outside which TestU01 reports a failure by default. We call *systematic* a failure that happens for all seeds. A more detailed discussion of this choice can be found in [Vigna 2014].

Note that when generating a floating-point number the lowest eleven bits returned by the generator *are not used at all* due to the fact that the mantissa of a 64-bit floating-point number is formed by 53 bits only. For this reason, we will consistently run our tests both on a generator and on its reverse.[5]

We applied a three-stage strategy using SmallCrush, Crush and BigCrush, which are increasingly stronger test suites from TestU01. We ran SmallCrush on all 272 full-period generators just found, isolating 141 which had less than 10 overall failures. We then ran Crush on the latter ones, and finally BigCrush on the top 8 results.

To get an intuition about the relative strength of the two techniques used to reduce linear artifacts (multiplication by a constant in `xorshift*` generators versus adding outputs in `xorshift+` generators), we also performed the same tests on `xorshift128*` generators, and run BigCrush on the 20 full-period triples for `xorshift1024+` generators reported in [Vigna 2014].

6. RESULTS

Figure 1 reports the score-rank plot of the distribution of SmallCrush combined failures for the 272 `xorshift128+` and `xorshift128*` generators. The plot associates with abscissa x the number of generators with x or more failures.[6]

While we cannot draw conclusions about a specific generator from this plot, the different decay of the two distributions suggests that the `xorshift*` approach yields a better improvement of the quality of the underlying `xorshift128` generator than the `xorshift+` approach (i.e., ultimately, the `XSadd` approach). Indeed, only 13 `xorshift128*` generators

[5] That is, on the generator obtained by reversing the order of the 64 bits returned.

[6] Score-rank plots are the numerosity-based discrete analogous of the complementary cumulative distribution function of scores. They give a much clearer picture than frequency dot plots when the data points are scattered and highly variable.

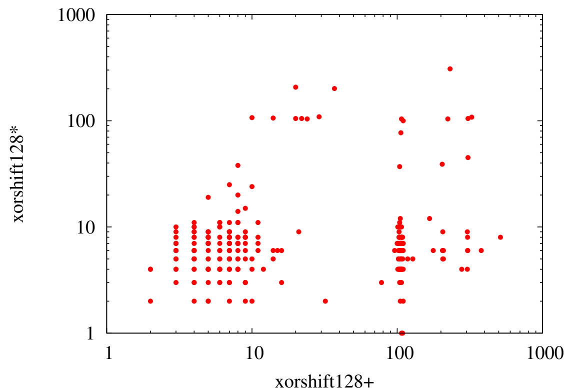

Fig. 2. Scatter plot of the combined SmallCrush scores for the 272 full-period `xorshift128+` and `xorshift128*` generators.

Table I. Results of BigCrush on the ten best `xorshift128+` generators following Crush.

a, b, c	Failures			W	Systematic
	S	R	+		
23, 17, 26	34	30	64	61	—
26, 19, 5	31	37	68	53	—
23, 18, 5	38	32	70	65	—
41, 11, 34	31	39	70	61	—
23, 31, 18	48	34	82	57	—
21, 23, 28	53	31	84	47	—
21, 16, 37	57	29	86	39	—
20, 21, 11	66	32	98	51	—
25, 8, 55	48	190	238	51	BirthdaySpacings
29, 13, 7	532	593	1125	57	RandomWalk1C, RandomWalk1H, RandomWalk1J, RandomWalk1M, RandomWalk1R

have systematic failures, against 104 `xorshift128+` generators, and the overall failures are 3711 against 16 171.

Another interesting fact is that the improvements to the quality of the generator (as measured by SmallCrush) due to the two methods are quite orthogonal. In Figure 2 we show the scatter plot of the combined SmallCrush scores for `xorshift128+` and `xorshift128*` generators: the only notable fact is that there are no good `xorshift128+` generator with an associated bad `xorshift128*` generator, while the converse happens. Kendall's τ between the two scores is 0.27, witnessing an almost complete lack of correlation.

Things get more complex using more stringent tests. In Table I we report the results of BigCrush on the ten best `xorshift128+` generators, and in Table II we report the same data for the 20 full-period generators identified in [Vigna 2014]. These tables should be compared with Table III, which report results for the ten best `xorshift128*` generators, and with Table IV, where we copy for completeness the results for `xorshift1024*` generators reported in [Vigna 2014].

Table II. Results of BigCrush on the `xorshift1024+` generators. The last five generators fail systematically a large number of tests.

a, b, c	Failures S	Failures R	Failures +	W
16, 23, 30	31	32	63	59
31, 11, 30	27	38	65	363
10, 11, 61	34	33	67	155
40, 11, 31	30	39	69	77
9, 14, 41	44	25	69	167
10, 9, 63	36	34	70	69
31, 33, 37	35	39	74	79
41, 7, 29	40	34	74	265
15, 16, 19	30	45	75	255
27, 13, 46	45	32	77	275
9, 5, 60	39	38	77	227
22, 7, 48	34	44	78	223
7, 16, 55	39	41	80	65
25, 8, 15	49	32	81	281
31, 10, 27	44	39	83	233
3, 26, 35	698	38	736	89
2, 11, 61	1108	34	1142	81
1, 13, 7	1521	46	1567	113
47, 1, 41	894	819	1713	99
51, 1, 46	890	1080	1970	111

Table III. Results of BigCrush on the ten best `xorshift128*` generators following Crush. All generators fail a MatrixRank test.

a, b, c	Failures S	Failures R	Failures +	W
26, 9, 27	128	124	252	29
17, 47, 29	131	126	257	27
13, 25, 19	129	130	259	51
49, 5, 26	134	128	262	63
49, 2, 25	128	135	263	43
40, 7, 27	141	129	270	47
28, 5, 33	140	131	271	39
16, 21, 1	143	132	275	65
44, 7, 18	133	153	286	53
16, 19, 22	144	143	287	45

Table IV. Results of BigCrush on the 20 full-period xorshift1024* generators. The last two generators fail systematically several tests.

a, b, c	Failures S	Failures R	Failures +	W
1, 13, 7	28	19	47	113
3, 26, 35	29	22	51	89
40, 11, 31	24	33	57	77
15, 16, 19	30	32	62	255
22, 7, 48	29	33	62	223
9, 14, 41	32	30	62	167
41, 7, 29	25	38	63	265
31, 11, 30	33	32	65	363
2, 11, 61	25	41	66	81
10, 11, 61	42	25	67	155
7, 16, 55	32	35	67	65
16, 23, 30	35	34	69	59
25, 8, 15	25	45	70	281
27, 13, 46	39	32	71	275
31, 10, 27	40	32	72	233
9, 5, 60	40	36	76	227
31, 33, 37	39	39	78	79
10, 9, 63	31	49	80	69
51, 1, 46	60	896	956	111
47, 1, 41	67	907	974	99

All xorshift128* generators fail the MatrixRank test: at this level of state-space size, multiplication is not able to hide such linear artifacts from BigCrush. On the other hand, among the best xorshift128+ generators selected by Crush some non-linear systematic failure appears, and, more importantly, in high dimension five, instead of two, full-period generators fail systematically several tests.

Table VI compares the BigCrush scores of the generators we discussed. For xorshift128+ we used the code of Figure 3. For xorshift128* we used the triple $49, 5, 26$ and for xorshift1024+/xorshift1024* the triple $31, 11, 30$.

6.1. Speed

Table V reports the speed of the generators discussed in the paper and of their xorshift* counterparts on an an Intel® Core™ i7-4770 CPU @3.40GHz (Haswell) using the gcc compiler (version 4.8.2). We measure the time that is required to emit 64 bits, so in the XSadd case we measure the time required to emit two 32-bit values.

The xorshift128+ case is particularly interesting because we can update the generator paying essentially no cost for the fact that the state space is more than 64 bits: as it is shown in Figure 3, we just need, while performing an update, to swap the role of the two 64-bit words of state space when we move them into temporary variables. The resulting code is incredibly tight, and, as it can be seen in Table V, gives rise to the fastest generator, also because we no longer need to manipulate the counter that would be necessary to update a xorshift1024+ generator.

A similar consideration can be made for a xorshift128* generator, which however is 32% slower—in fact, slower than a xorshift1024* generator. Certainly this difference is due in

Table V. Time to emit a 64-bit integer on an Intel® Core™ i7-4770 CPU @3.40GHz (Haswell).

Algorithm	Speed (ns/64 bits)
xorshift128+	1.10
xorshift1024+	1.34
xorshift64*	1.60
xorshift128*	1.45
xorshift1024*	1.36
xorshift4096*	1.36
xorgens4096	1.81
XSadd	1.98

```
#include <stdint.h>

uint64_t s[ 2 ];

uint64_t next(void) {
  uint64_t s1 = s[ 0 ];
  const uint64_t s0 = s[ 1 ];
  s[ 0 ] = s0;
  s1 ^= s1 << 23; // a
  return ( s[ 1 ] = ( s1 ^ s0 ^ ( s1 >> 17 ) ^ ( s0 >> 26 ) ) ) + s0; // b, c
}
```

Fig. 3. The suggested `xorshift128+` generator in C99 code. The array `s` should be initialized to a nonzero seed before calling `next()`.

Table VI. A comparison of generators using BigCrush.

Algorithm	Failures S	Failures R	Failures +	W	Systematic
xorshift128+	34	30	64	61	—
XSadd	38	850	888	13	LinearComp, MatrixRank, MaxOft, Permutation
xorshift64*	230	133	363	31	MatrixRank
xorshift128*	134	128	262	63	MatrixRank
xorshift1024*	29	22	51	363	—
xorshift1024+	27	38	65	363	—
xorshift4096*	33	34	67	441	—
xorgens4096	42	40	82	961	—
java.util.Random	4078	9486	13564	—	Almost all

part to compiler and CPU artifacts, and in part to the quirky nature of microbenchmarking, as the difference between `xorshift1024*` and `xorshift1024+` is barely detectable; but the difference is nonetheless significant. In particular, a Java implementation of a `xorshift128+` is particularly efficient as it does not require access to an array (a particularly slow operation in Java), and it provides a high-quality substitute that is faster than the dreaded generator provided by the language.

Table VII. Mean and variance for the data shown in Figure 4.

Algorithm	Mean	Variance
xorshift64*	0.5005	0.0038
xorshift128*	0.4996	0.0048
xorshift128+	0.4970	0.0288
XSadd	0.4957	0.0302
xorshift1024*	0.4918	0.0327
xorshift1024+	0.4575	0.1046
xorshift4096*	0.4256	0.0805

6.2. Equidistribution

It is known that a `xorshift` generator with a state space of n bits is $n/64$-dimensionally equidistributed, and that the associated `xorshift*` generator inherits this property [Vigna 2014]. It is easy to show that a slightly weaker property is true of the associated `xorshift+` generator:

PROPOSITION 6.1. *If a `xorshift` generator is k-dimensionally equidistributed, the associated `xorshift+` generator if $(k-1)$-dimensionally equidistributed.*

PROOF. Consider a $(k-1)$-tuple $\langle t_1, t_2, \ldots, t_{k-1}\rangle$. For each possible value x_0, there is exactly one k-tuple $\langle x_0, x_1, \ldots, x_{k-1}\rangle$ such that $x_{i-1} + x_i = t_i$ (the sum is in $\mathbf{Z}/2^{64}\mathbf{Z}$), for $0 < i < k$. Thus, there are exactly 2^{64} appearances of the $(k-1)$-tuple $\langle t_1, t_2, \ldots, t_{k-1}\rangle$ in the sequence emitted by a `xorshift+` generator associated with a k-dimensionally equidistributed `xorshift` generator, with the exception of the zero tuple, for which the appearance associated with the zero k-tuple is missing. □

Note that in general it is impossible to claim k-dimensional equidistribution. Consider the full-period 6-bit generator that uses 3-bit shifts with $a = 1$, $b = 2$ and $c = 1$. As a `xorshift` generator with a 3-bit output (the lowest bits), it is 2-dimensionally equidistributed. However, it is easy to verify that the sequence of outputs of the associated `xorshift+` generator contains twice the pair of consecutive 3-bit values $\langle 000, 000\rangle$, so the generator is 1-, but not 2-dimensionally equidistributed.

6.3. Escaping zeroland

We show in Figure 4 the speed at which the generators hitherto examined "escape from zeroland" [Panneton et al. 2006]: purely linearly recurrent generators with a very large state space need a very long time to get from an initial state with a small number of ones to a state in which the ones are approximately half. The figure shows a measure of escape time given by the ratio of ones in a window of 4 consecutive 64-bit values sliding over the first 1000 generated values, averaged over all possible seeds with exactly one bit set (see [Panneton et al. 2006] for a detailed description). Table VII condenses Figure 4 into the mean and variance of the displayed values.

There are three clearly defined blocks: `xorshift64*` and `xorshift128*`; then, `XSadd`, `xorshift128+` and `xorshift1024*`; and finally, `xorshift1024+` and `xorshift4096*`. These blocks are reflected also in the mean and variance data reported in Table VII. The clear conclusion is that the `xorshift*` approach yields generators with better behavior.

7. CONCLUSIONS

In this paper we discussed the family of `xorshift+` generators—a variant of `XSadd` based on 64-bit shifts. On modern processors, our suggested `xorshift128+` generator is currently

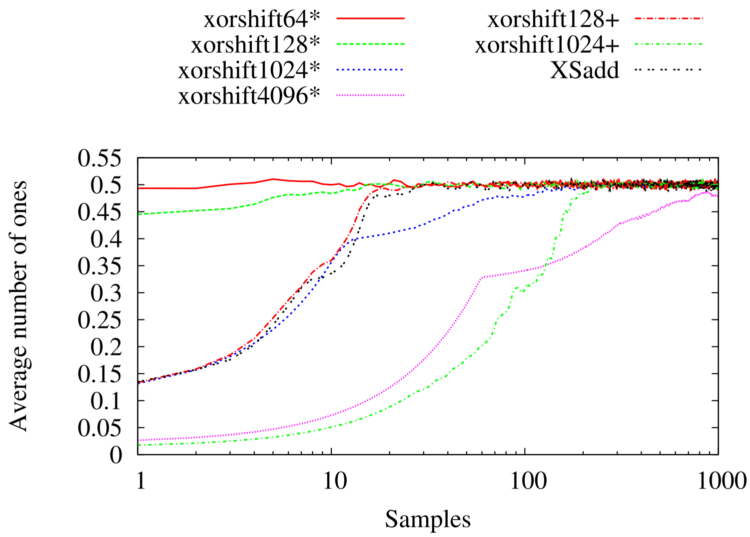

Fig. 4. Convergence to "half of the bits are ones in average" plot.

the fastest full-period generator that does not fail systematically any BigCrush test (not even reversed).

Higher dimension `xorshift+` generators are less interesting. The speed gain with respect to a `xorshift*` generator with the same state space is almost undetectable, and the theoretical properties we can prove are weaker: every bit of the output of a full-period `xorshift*` generator has full period [Vigna 2014], but proving the same for a `xorshift+` generator appears to be more difficult because of the dependencies induced by carry propagation. Also the equidistribution properties that are preserved by permuting the output with a multiplication (i.e., an n-bit `xorshift*` generator is $n/64$-dimensionally equidistributed—each $n/64$-tuple of consecutive 64-bit values is output exactly once, except for a missing tuple of zeroes) weaken in a `xorshift+` generator, as discussed in Section 6.2.

Empirical analysis of full-period high-dimensional generators studied in this paper suggests also that in high dimension the `xorshift*` approach yields a better improvement to the quality of the underlying `xorshift` generator.

Finally, `xorshift+` generators have a worse behavior when started from a state containing a large fraction of zeroes—the number of ones in the output moves back to normality after a number of steps comparable with that of `xorshift*` generators with a larger state space.

We conclude that while a good `xorshift128+` generator (Figure 3) is an excellent drop-in substitute for the low-dimensional generators found in many programming languages, in high dimension the `xorshift*` approach appears to yield better generators.

REFERENCES

Richard P. Brent. 2004. Note on Marsaglia's Xorshift Random Number Generators. *Journal of Statistical Software* 11, 5 (2004), 1–5.

Richard P. Brent. 2007. Some long-period random number generators using shifts and xors. In *Proceedings of the 13th Biennial Computational Techniques and Applications Conference, CTAC-2006 (ANZIAM J.)*, Wayne Read and A. J. Roberts (Eds.), Vol. 48. C188–C202.

Aaldert Compagner. 1991. The hierarchy of correlations in random binary sequences. *Journal of Statistical Physics* 63, 5-6 (1991), 883–896.

Pierre L'Ecuyer and Richard Simard. 2007. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.* 33, Article 22 (August 2007). Issue 4.

Rudolf Lidl and Harald Niederreiter. 1994. *Introduction to finite fields and their applications*. Cambridge University Press, Cambridge.

George Marsaglia. 2003. Xorshift RNGs. *Journal of Statistical Software* 8, 14 (2003), 1–6.

Harald Niederreiter. 1992. *Random number generation and quasi-Monte Carlo methods*. CBMS-NSF regional conference series in Appl. Math., Vol. 63. SIAM.

François Panneton and Pierre L'Ecuyer. 2005. On the xorshift random number generators. *ACM Trans. Model. Comput. Simul* 15, 4 (2005), 346–361.

François Panneton, Pierre L'Ecuyer, and Makoto Matsumoto. 2006. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Softw.* 32, 1 (2006), 1–16.

Mutsuo Saito and Makoto Matsumoto. 2014. XSadd (Version 1.0). (2014). Retrieved March 11, 2014 from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/XSADD/

Sebastiano Vigna. 2014. An experimental exploration of Marsaglia's `xorshift` generators, scrambled. *CoRR* abs/1402.6246 (2014).