Further scramblings of Marsaglia’s xorshift generators

Sebastiano Vigna

Università degli Studi di Milano, Italy

Abstract

xorshift* generators are a variant of Marsaglia’s xorshift generators that eliminate linear artifacts typical of generators based on $\mathbb{Z}/2\mathbb{Z}$-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 $\mathbb{Z}/2^{32}\mathbb{Z}$, leading to the XSadd generator. Starting from the observation that the lower bits of XSadd are very weak, as its reverse fails several statistical tests, we explore variants of XSadd using 64-bit operations, and describe in detail xorshift128+, an extremely fast generator that passes strong statistical tests using only three shifts, four xors and an addition.

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].

Shortly after the introduction of high-dimensional xorshift* generators, Saito and Matsumoto [2014] proposed a different way to 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 $\mathbb{Z}/2^{32}\mathbb{Z}$) with the previous output. Since the sum in $\mathbb{Z}/2^{32}\mathbb{Z}$ is not linear over $\mathbb{Z}/2\mathbb{Z}$, the result should be free of linear artifacts.

Their generator XSadd has 128 bits of state and full period $2^{128} - 1$. However, while XSadd passes BigCrush, its reverse fails the LinearComp, MatrixRank, MaxOft and Permutation test of BigCrush, which highlights a significant weakness in its lower bits.

*This work has been supported by a Google Focused Research Award.
In this paper, leveraging the theoretical and experimental data about xorshift generators contained in (Vigna, 2016), 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 any test from the BigCrush suite of TestU01 (even reversed) and generates 64 pseudorandom bits in 1.06 ns on an Intel® Core™ i7-4770 CPU @3.40GHz (Haswell). It is the fastest full-period generator we are aware of with such empirical statistical properties, making it an excellent drop-in substitute for the low-dimensional generators found in many programming languages.

Indeed, Google has recently chosen to use xorshift128+ as the PRNG of its Javascript engine V8, which powers the Chrome browser. Immediately after, Firefox and Safari made the same choice, making xorshift128+ one of the most widely deployed PRNGs.

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.

2. xorshift generators

The basic idea of xorshift generators is that the state is modified by applying repeatedly a shift and an exclusive-or (xor) operation. In this paper we consider 64-bit shifts and states made of $2^n$ bits, with $n \geq 7$. We usually append $n$ to the name of a family of generators when we need to restrict the discussion to a specific state size.

In linear-algebra terms, if $L$ is the $64 \times 64$ matrix on $\mathbb{Z}/2\mathbb{Z}$ that effects a left shift of one position on a binary row 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 $(I + L^s)$ or $(I + R^s)$, respectively, where $s$ is the amount of shifting.

As suggested by Marsaglia (2003), we use always three low-dimensional 64-bit shifts, but locating them in the context of a larger block matrix of the form:

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

1http://v8project.blogspot.it/2015/12/theres-mathrandom-and-then-theres.html
2http://prng.di.unimi.it/
3A more detailed study of the linear algebra behind xorshift generators can be found in Marsaglia (2003) and Panneton and L’Ecuyer (2005).
4We remark that XSadd uses a slightly different matrix, in which the bottom right element is $1 + L^c$. 

2
It is useful to associate with a linear transformation $M$ its characteristic polynomial
\[ P(x) = \det(M - xI). \]
The associated generator has maximum-length period if and only if $P(x)$ is primitive over $\mathbb{Z}/2\mathbb{Z}$. This happens if $P(x)$ is irreducible and if $x$ has maximum period in the ring of polynomial over $\mathbb{Z}/2\mathbb{Z}$ modulo $P(x)$.

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, 2016), 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 as a further simplification of the Weyl generator idea: instead of keeping track of a separate generator, XSadd adds (in $\mathbb{Z}/2^{32}\mathbb{Z}$) consecutive outputs of an underlying xorshift generator. In this way, we introduce a nonlinear operation without enlarging the state. In practice, this amounts to returning the sum of the currently updated word and of the last updated word of the state.

Saito and Matsumoto (2014) claim that XSadd does not fail 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 a number of tests, including some that are not due to linear artifacts, suggesting that its latter 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, 2016).

#### 3.1. Equidistribution and full period

In our context, a generator with $n$ bits of state and $t$ output bits is $k$-dimensionally equidistributed if over the whole output every $k$-tuple of consecutive output values appears $2^{n-t-k}$ times, except for the zero $k$-tuple, which appears $2^{n-t-k} - 1$ times. It is known that a xorshift generator with a state of $n$ bits is $n/64$-dimensionally equidistributed, and that the associated xorshift\* generator inherits this property (Vigna, 2016). It is easy to show that a slightly weaker property is true of the associated xorshift+ generator:
Proposition 1. If a xorshift generator is \(k\)-dimensionally equidistributed, the associated xorshift+ generator is \((k-1)\)-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 \(\mathbb{Z}/2^{64}\mathbb{Z}\)), 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 \((k-1)\)-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.

An immediate consequence is that every individual bit of the generator (and thus a fortiori the entire output) has full period. We will need the following result, which is Proposition 7.1 from Vigna (2016):

Proposition 2. Let \(x_0, x_1, \ldots, x_{2^n-2}\) be a list of \(2^t\)-bit values, \(t \leq n\), such that every value appears \(2^{n-t}\) times, except for 0, which appears \(2^{n-t}-1\) times. Then, for every fixed bit \(k\) the associated sequence has period \(2^n - 1\).

Proposition 3. Every bit of a xorshift+ generator with \(n\) bits of state has period \(2^n - 1\).

Proof. Since \(n \geq 7\), by Proposition 1 a xorshift+ generator is at least 1-dimensionally equidistributed, and we just have to apply Proposition 7.1 from Vigna (2016).

We remark that, similarly to a xorshift or xorshift* generator, the lowest bit of a xorshift+ generator satisfies a linear recurrence, as on the lowest bit the effect of an addition is the same as that of a xor.

3.2. Choosing the shifts

Vigna (2016) provides choices of shifts for full-period generators with 1024 or 4096 bits of state. In this paper, however, we want to explore the idea of xorshift+ generators with 128 bits of state to provide an alternative to XSadd that is free of its statistical flaws, and faster on modern 64-bit CPUs. Finding generators with a small state space, strong statistical properties and

\footnote{It should be remarked that at least the two lowest bits of a xorshift+ generator satisfy a linear recurrence; they become three if the multiplier is congruent to 1 modulo 4, as it happens in Vigna (2016).}
speed comparable with that of a linear congruential generator is an interesting practical 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 then ran experiments following the protocol used in [Vigna 2016], 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 state we use the seeds $1 + i \lfloor 2^{64} / 100 \rfloor$, $0 \leq i < 100$. 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 $[0.001 \ldots 0.999]$. 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 2016]. Note that we run our tests both on a generator and on its reverse, that is, on the generator obtained by reversing the order of the 64 bits returned. The final score is the sum of the number of tests failed by a generator and its reverse.

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 described above, isolating 141 which had less than 10 overall failures. We then ran Crush on the latter ones, and finally BigCrush on the top 10 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 ran BigCrush on the 20 full-period triples for xorshift1024+ generators reported in [Vigna 2016].

4. Results

In Table 1 we report the results of BigCrush on the ten best xorshift128+ generators: we show the number of failures of a generator, of its reverse, their sum, the weight of the associated polynomial and, finally, systematic failures, if any; it should be compared with Table 3 which report results for the ten best xorshift128* generators. In Table 2 we report the same data for the 20
Table 1: Results of BigCrush on the ten best xorshift128+ generators following Crush.

<table>
<thead>
<tr>
<th>a, b, c</th>
<th>Failures</th>
<th>Weight</th>
<th>Systematic failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>23, 17, 26</td>
<td>34 30 64</td>
<td>61</td>
<td>—</td>
</tr>
<tr>
<td>26, 19, 5</td>
<td>31 37 68</td>
<td>53</td>
<td>—</td>
</tr>
<tr>
<td>23, 18, 5</td>
<td>38 32 70</td>
<td>65</td>
<td>—</td>
</tr>
<tr>
<td>41, 11, 34</td>
<td>31 39 70</td>
<td>61</td>
<td>—</td>
</tr>
<tr>
<td>23, 31, 18</td>
<td>48 34 82</td>
<td>57</td>
<td>—</td>
</tr>
<tr>
<td>21, 23, 28</td>
<td>53 31 84</td>
<td>47</td>
<td>—</td>
</tr>
<tr>
<td>21, 16, 37</td>
<td>57 29 86</td>
<td>39</td>
<td>—</td>
</tr>
<tr>
<td>20, 21, 11</td>
<td>66 32 98</td>
<td>51</td>
<td>—</td>
</tr>
<tr>
<td>25, 8, 55</td>
<td>48 190 238</td>
<td>51</td>
<td>BirthdaySpacings</td>
</tr>
<tr>
<td>29, 13, 7</td>
<td>532 593 1125</td>
<td>57</td>
<td>RandomWalk1C, RandomWalk1H, RandomWalk1J, RandomWalk1M, RandomWalk1R</td>
</tr>
</tbody>
</table>

full-period generators identified in (Vigna 2016), which should be compared with Table VI therein.

All xorshift128* generators fail the MatrixRank test when reversed: with this state size, multiplication is not able to hide such linear artifacts from BigCrush, as the two lowest bit of such generators satisfy a linear recurrence. On the other hand, among the best xorshift128+ generators selected by Crush some non-linear systematic failure appears.

Table 4 compares the BigCrush scores of the generators we discussed. For xorshift128+ we used the triple 23, 18, 5 (Figure 1). For xorshift128* we used the triple 17, 19, 30 and for xorshift1024+/xorshift1024* the triple 31, 11, 30 (the xorshift1024* generator is the one proposed in (Vigna 2016)).

Our choice of triples is based not only on the BigCrush scores and on polynomial weight, but also on an additional datum: the result of POP (“p-value of p-values”) tests. BigCrush generates 254 p-values, each corresponding to a specific statistics (the same test might generate several statistics). If the source is perfectly random, and the statistics distribution is known exactly, the p-values generated at different points of the state space should appear to be uniformly distributed. We can thus test whether this is true for each one of the 254 gener-
Table 2: Results of BigCrush on the xorshift1024+ generators. The last five generators fail systematically a large number of tests.

<table>
<thead>
<tr>
<th>a, b, c</th>
<th>Failures</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S</td>
<td>R</td>
</tr>
<tr>
<td>16, 23, 30</td>
<td>29</td>
<td>26</td>
</tr>
<tr>
<td>31, 11, 30</td>
<td>34</td>
<td>29</td>
</tr>
<tr>
<td>27, 13, 46</td>
<td>30</td>
<td>34</td>
</tr>
<tr>
<td>9, 14, 41</td>
<td>29</td>
<td>36</td>
</tr>
<tr>
<td>10, 11, 61</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td>25, 8, 15</td>
<td>28</td>
<td>42</td>
</tr>
<tr>
<td>40, 11, 31</td>
<td>32</td>
<td>39</td>
</tr>
<tr>
<td>7, 16, 55</td>
<td>43</td>
<td>28</td>
</tr>
<tr>
<td>15, 16, 19</td>
<td>38</td>
<td>34</td>
</tr>
<tr>
<td>31, 33, 37</td>
<td>39</td>
<td>35</td>
</tr>
<tr>
<td>9, 5, 60</td>
<td>39</td>
<td>35</td>
</tr>
<tr>
<td>22, 7, 48</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>10, 9, 63</td>
<td>38</td>
<td>45</td>
</tr>
<tr>
<td>31, 10, 27</td>
<td>43</td>
<td>40</td>
</tr>
<tr>
<td>41, 7, 29</td>
<td>50</td>
<td>53</td>
</tr>
<tr>
<td>3, 26, 35</td>
<td>1014</td>
<td>29</td>
</tr>
<tr>
<td>2, 11, 61</td>
<td>1040</td>
<td>41</td>
</tr>
<tr>
<td>1, 13, 7</td>
<td>1332</td>
<td>35</td>
</tr>
<tr>
<td>47, 1, 41</td>
<td>819</td>
<td>777</td>
</tr>
<tr>
<td>51, 1, 46</td>
<td>844</td>
<td>1047</td>
</tr>
</tbody>
</table>
Table 3: Results of BigCrush on the ten best xorshift128* generators following Crush. All generators fail a MatrixRank test when reversed.

<table>
<thead>
<tr>
<th>a, b, c</th>
<th>Failures</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S  R   +</td>
<td></td>
</tr>
<tr>
<td>49, 2, 25</td>
<td>28 130 158</td>
<td>43</td>
</tr>
<tr>
<td>13, 15, 38</td>
<td>32 131 163</td>
<td>47</td>
</tr>
<tr>
<td>20, 21, 31</td>
<td>38 129 167</td>
<td>37</td>
</tr>
<tr>
<td>44, 7, 18</td>
<td>29 140 169</td>
<td>53</td>
</tr>
<tr>
<td>13, 15, 53</td>
<td>39 132 171</td>
<td>47</td>
</tr>
<tr>
<td>36, 23, 29</td>
<td>39 133 172</td>
<td>53</td>
</tr>
<tr>
<td>10, 19, 15</td>
<td>40 140 180</td>
<td>45</td>
</tr>
<tr>
<td>31, 33, 18</td>
<td>32 149 181</td>
<td>47</td>
</tr>
<tr>
<td>17, 19, 30</td>
<td>36 146 182</td>
<td>61</td>
</tr>
<tr>
<td>22, 5, 16</td>
<td>32 222 254</td>
<td>57</td>
</tr>
</tbody>
</table>

ated values using a goodness-of-fit test to get a p-value (which is a p-value of p-values): NIST [Rukhin et al. 2001] suggests the threshold 10^{-4} on a \chi^2 test on the counts of the p-values falling in the intervals [k/10 . . . (k+1)/10], 0 \leq k < 10; we used the more stringent value 10^{-3} on a Kolmogorov-Smirnov test for the uniform (continuous) distribution. The triples we suggest for xorshift+ do not fail any POP test, and the same happens for the xorshift1024* generator suggested in [Vigna 2016], but, for example, the first triple listed in Table 1 fails four POP tests.

5. Jumping ahead

The simple form of a xorshift generator makes it trivial to jump ahead quickly by any number of next-state steps. If v is the current state, we want to compute \( vM^j \) for some j. But \( M^j \) is always expressible as a polynomial in \( M \) of degree lesser than that of the characteristic polynomial. To find such a polynomial it suffices to compute \( x^j \mod P(x) \), where \( P(x) \) is the characteristic polynomial of \( M \). Such a computation can be easily carried out using standard techniques (quadratures to find \( x^2 \mod P(x) \), etc.), leaving us with

---

8Actually, four p-values (two from the LongestHeadRun test and two from the Fourier3 test) have been dropped as they are based on rather approximate computation of the p-value of the statistics, as documented by the authors of TestU01, and thus tend to generate spurious errors.
Figure 1: The xorshift128+ generator used in the tests.

```c
#include <stdint.h>

uint64_t s[2];

uint64_t next(void) {
    uint64_t s1 = s[0];
    const uint64_t s0 = s[1];
    const uint64_t result = s0 + s1;
    s[0] = s0;
    s1 ^= s1 << 23; // a
    s[1] = s1 ^ s0 ^ (s1 >> 18) ^ (s0 >> 5); // b, c
    return result;
}
```

A polynomial $Q(x)$ such that $Q(M) = M^j$. Now, if

$$Q(x) = \sum_{i=0}^{n} \alpha_i x^i,$$

we have

$$vM^j = vQ(M) = \sum_{i=0}^{n} \alpha_i vM^i,$$

and now $vM^i$ is just the $i$-th state after the current one. If we known in advance the $\alpha_i$'s, computing $vM^j$ requires just computing the next state for $n$ times, accumulating by xor the $i$-th state iff $\alpha_i \neq 0$.

In general, one needs to compute the $\alpha_i$'s for each desired $j$, but the practical usage of this technique is that of providing subsequences that are guaranteed to be non-overlapping. We can fix a reasonable jump, for example $2^{64}$ for a xorshift128+ generator, and store the $\alpha_i$’s for such a jump as a bit mask. Operating the jump is now entirely trivial, as it requires at most 128 state changes. In Figure 3 we show the jump function for the generator of Figure 1. By iterating the jump function, one can access $2^{64}$ non-overlapping sequences of length $2^{64}$ (except for the last one, which will be of length $2^{64} - 1$).

5.1. Speed

Table 4 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.

9Brent’s ranut generator [Brent 1992] contains one of the first applications of this technique.
Figure 2: The xorshift1024+ generator used in the tests.

#include <stdint.h>

uint64_t s[16];
int p;

uint64_t next(void) {
    const uint64_t s0 = s[p];
    uint64_t s1 = s[p = (p + 1) & 15];
    const uint64_t result = s0 + s1;
    s1 ^= s1 << 31; // a
    s[p] = s1 ^ s0 ^ (s1 >> 11) ^ (s0 >> 30); // b, c
    return result;
}

(Haswell). We measured 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. We used suitable options to keep the compiler from unrolling loops or extracting loop invariants.

The xorshift128+ case is particularly interesting because we can update the generator paying essentially no cost for the fact that the state is made of more than 64 bits: as it is shown in Figure 1 we just need, while performing an update, to swap the role of the two 64-bit words of state when we move them into temporary variables. The resulting code is incredibly tight, and, as it can be seen in Table 4, 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).

5.2. 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 5 condenses Figure 4 into the mean and standard deviation of the displayed values.

There are three clearly defined blocks: xorshift128*; then, XSadd, xorshift128* and xorshift1024*; finally, xorshift1024+. These blocks are reflected also in
#include <stdint.h>

void jump(void) {
    static const uint64_t JUMP[] = { 0x8a5cd789635d2dff,
     0x121fd2155c472f96 };

    uint64_t s0 = 0;
    uint64_t s1 = 0;
    for(int i = 0; i < sizeof JUMP / sizeof *JUMP; i++)
        for(int b = 0; b < 64; b++) {
            if (JUMP[i] & 1ULL << b) {
                s0 ^= s[0];
                s1 ^= s[1];
            }
            next();
        }
    s[0] = s0;
    s[1] = s1;
}

Table 5. The clear conclusion is that the xorshift* approach yields generators with faster escape.

6. Conclusions

We discussed the family of xorshift+ generators—a variant of XSadd based on 64-bit shifts. In particular, we described a xorshift128+ generator that is currently the fastest full-period generator we are aware of that does not fail systematically any BigCrush test (not even reversed). xorshift128+ can be easily implemented in hardware, as it requires just three shift, four xors and an addition, and have been widely deployed as the basic random generator in Javascript engine of all major browsers.

Higher-dimensional xorshift+ generators “escape from zeroland” too slowly, making them less interesting than their xorshift* counterpart.

Brent, R.P., 1992. Uniform random number generators for supercomputers, in: Supercomputing, the competitive advantage: proceedings of the Fifth
Table 4: A comparison of generators.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Speed (ns/64 b)</th>
<th>Failures</th>
<th>W/n</th>
<th>Systematic failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>xorshift128+</td>
<td>1.06</td>
<td>38</td>
<td>32</td>
<td>70</td>
</tr>
<tr>
<td>xorshift128*</td>
<td>1.18</td>
<td>36</td>
<td>146</td>
<td>182</td>
</tr>
<tr>
<td>xorshift1024+</td>
<td>1.32</td>
<td>34</td>
<td>29</td>
<td>63</td>
</tr>
<tr>
<td>xorshift1024*</td>
<td>1.34</td>
<td>33</td>
<td>32</td>
<td>65</td>
</tr>
<tr>
<td>XSadd</td>
<td>2.06</td>
<td>38</td>
<td>850</td>
<td>888</td>
</tr>
</tbody>
</table>

MatrixRank, 
MaxOft, 
Permutation

Figure 4: Convergence to “half of the bits are ones in average” plot.

Table 5: Mean and standard deviation for the data shown in Figure 4.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mean</th>
<th>Standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>xorshift128*</td>
<td>0.4994</td>
<td>0.0047</td>
</tr>
<tr>
<td>xorshift128+</td>
<td>0.4974</td>
<td>0.0239</td>
</tr>
<tr>
<td>XSadd</td>
<td>0.4957</td>
<td>0.0302</td>
</tr>
<tr>
<td>xorshift1024*</td>
<td>0.4935</td>
<td>0.0296</td>
</tr>
<tr>
<td>xorshift1024+</td>
<td>0.4575</td>
<td>0.1045</td>
</tr>
</tbody>
</table>


