A Fast Hardware Pseudorandom Number Generator Based on the xoroshiro128 LFSR

James Hanlon, Diya Rajan, and Stephen Felix

Abstract—The Graphcore Intelligent Processing Unit (IPU) contains an original pseudorandom number generator (PRNG) called xoroshiro128aox that is based on the xoroshiro128 LFSR. It is designed to be cheap to implement in hardware and provide a high quality of statistical randomness. In this paper, we present a rigorous assessment of the quality of our new PRNG using standard statistical test suites and compare the results with the fast contemporary PRNGs xoroshiro128+, pcg64 and philox4x32. As a baseline for the analysis, we include the widely-used Mersenne Twister PRNG. In our experiments, we show that xoroshiro128aox mitigates the known weakness in the lower order bits of xoroshiro128+ with our new AOX output function by passing the BigCrush and PractRand test suites. We extend our testing with the Gjrand test suite and a Hamming-Weight dependency test to highlight the linear weaknesses of both xoroshiro128 PRNGs, but conclude that these linearities are hard to detect, and the xoroshiro128aox PRNG otherwise provides an excellent trade off between statistical quality and hardware implementation cost.

1 INTRODUCTION

Randomness is widely used in machine intelligence (MI) algorithms, with examples including shuffling of data prior to each training epoch for stochastic gradient descent [1], sub sampling of training images, weight initialisation, adding noise to activations or weights, regularisation techniques like Dropout [2] and Stochastic Pooling [3], Monte Carlo sampling in generative models, choosing random actions in reinforcement-learning models, or random selection of directions in the weight space during training [4]. The use of pseudorandom number generators (PRNGs), which are deterministic algorithms for generating sequences of numbers that appear to be random, are ubiquitous in computing. When compared with true random number generators (TRNGs) that are typically based on sampling of some physical phenomena such as ring oscillators [5], PRNGs offer a higher rate of output, no requirement for special hardware structures, and to have the ability to be seeded to replay a sequence deterministically. In application domains such as AI, there is no need for a PRNG to be cryptographically secure, meaning that it does not need to be difficult for an adversary to predict future outputs based on past ones.

It is not clear to what degree the statistical quality of a PRNG affects the performance of MI applications. For weight initialisation, a minor bias may have a negligible impact since backpropagation tunes these weights and should lead to locally optimum solutions regardless of whether the weights were perfectly randomly distributed initially. On the other hand, applications such as Monte-Carlo approximations could produce biased, inaccurate solutions. It is interesting to note that Python’s default PRNG is the 32-bit Mersenne Twister MT19937 algorithm\(^1\), which fails standard statistical tests [6]. This situation suggests that high-quality randomness may not be crucial to all aspects of MI applications, or indeed more widely in other application areas.

Software PRNGs can be implemented with a very low overhead of just a few instructions per output. However, when randomness is required more frequently, generation in hardware can provide performance that is orders of magnitude better than compatible generation in software. In the Graphcore Intelligent Processing Unit (IPU) [7], each of its 1,216 tile processors contains a novel PRNG called xoroshiro128aox that is capable of producing 64 bits of random data every cycle. This randomness is used either to automatically round floating-point numbers stochastically [8] or is made available to the programmer through instructions to generate random values in uniform and Gaussian distributions.

This paper presents a rigorous assessment of the quality of the IPU’s PRNG using standard statistical test suites. Our results indicate that its quality is comparable to contemporary fast non-cryptographic PRNGs with comparable state size, whilst being cheap to implement. The remainder of this paper is structured as follows. Section 2 describes standard PRNG statistical testing, and the specific test suites that are used in our investigation. Section 3 introduces the IPU’s PRNG and the xoroshiro128 family that it is derived from. Section 5 describes the methodology used to perform the statistical tests, as well as the other generators included for comparison. Section 6 presents the results of the empirical statistical analysis. Section 7 analyses the hardware implementation cost of the generators considered by synthesising them in hardware. Section 8 identifies several other aspects of PRNG quality and analyses these for xoroshiro128aox. Section 9 concludes the investigation.

2 STATISTICAL TESTING

Theoretical analysis of a PRNG can be used to establish some properties, such as that values are produced uniformly and over their entire period length, however only empirical testing can be used to establish the statistical properties of a PRNG, and is the standard approach for judging quality. An empirical statistical testing involves sampling the output of

---

\(^1\) See Python 3’s random module documentation: https://docs.python.org/3/library/random.html
a generator, calculating some kind of summary statistic such as mean or standard deviation, then comparing this to the same statistic for a truly random source. This approach is only applicable when the number of samples is less than the sequence length, since it would otherwise be easy to detect a repeating sequence.

Empirical testing is formalised by testing against a null hypothesis, where we assume the the output of the PRNG follows a discrete uniform distribution. Probabilities called \( p \)-values are calculated from a test statistic such that they indicate how likely a result is assuming the null hypothesis to be true. An extreme \( p \)-value therefore indicates an output or set of outputs that are unlikely to be random. A truly random source of data produces \( p \)-values according to a Poisson distribution between 0 and 1, and so \( p \)-values very close to 0 or 1 are unlikely given the null hypothesis. In statistical testing of PRNGs, results may be categorised such that extreme \( p \)-values will be flagged as ‘suspicious’ or ‘anomalous’, but to determine a pass or fail result, an arbitrary threshold can be applied to summary \( p \)-values. The exact bounds may depend on the test.

Because there are an infinite number of statistical tests that can be devised, and each test will explore different aspects of the PRNG, it is not possible for any set of tests to perfectly determine whether a generator is perfectly random. Indeed, since we are interested in building a generator that we know is inherently non random, we are concerned only that the generator is good: that it passes all simple tests and fails complex tests infrequently [9].

For our empirical analysis, we use the statistical test suites for RNGs provided by the TestU01 [10], PractRand [11] and Gjrand [12] libraries. These are all well regarded by the community for their ability to distinguish good from bad PRNGs. Notable other test suites are DieHarder, RaBiGeTe and NIST STS, but are less comprehensive, less well regarded by the community and less well maintained.

TestU01 provides implementations of standard statistical tests such as Birthday Spacings that measures the distribution of distances between outputs and Collision that measures the probability of outputs occurring in the same interval of bits multiple times. Sets of statistical tests are instanced in test batteries that can be used for exercising RNGs. The most stringent battery BigCrush contains 160 test parameterisations using 30 individual tests, and consumes close to 1 TB of random data. The tests conducted and the parameters used are explained in detail in the TestU01 User Guide [9]. The BigCrush test battery runs 106 individual tests, with some tests run with different configurations, using close to 1 TB of random numbers. Each test can produce one or more \( p \)-values, and in total a single run of BigCrush produces 160 \( p \)-values. In total, 254 \( p \)-values are reported in the output of BigCrush, however, only independent \( p \)-values are reported in the summary, which we count towards failures in our analysis.

PractRand (Practically Random) provides statistical testing of RNGs using five tests\(^2\), deployed in many different parameterisations. Several of these tests are original and developed by the software’s author Chris Doty-Humphrey, making it a complement to TestU01’s BigCrush. PractRand provides the ability for generators to be tested with effectively unlimited sequence lengths, although the default sequence length is 32 TB, which we use in our analysis. For this test length, PractRand runs 455 test parameterisations of the five individual tests.

Gjrand is another library of PRNGs and statistical tests, created by David Blackman, a co-creator of the xoroshiro128 family of generators. Its test suite for uniform random bits performs 13 tests and is limited to a maximum of 10 TB of output, which we use for our analysis.

We also include the Hamming-weight dependency (HWD) test used in the xoroshiro paper [13], which looks for dependencies between the numbers of ones and zeros in consecutive outputs, which is indicative of generators based on LFSRs. The HWD test is based on the \( z^9 \) test in Gjrand \(^3\). The test only counts set bits and does not interpret the bit patterns as values. Note that the aforementioned test sets include similar tests to HWD that count the frequency of set bits: linear-complexity LinearComp of BigCrush and DC6, and BCFN of PractRand.

3 XOROSHIRO128AOX

The IPU’s PRNG is based on the xoroshiro128 linear-feedback shift register (LFSR), developed by Blackman and Vigna in 2017 [13]. This LFSR operates by performing the operations exclusive or (XOR), rotate, shift and rotate consecutively on 128 bits of state. Because successive outputs produced by LFSRs are correlated, they are easy to detect using statistical tests such as auto correlation. A family of more robust PRNGs is obtained by adding a non-linear function of the state vector to ‘scramble’ the output. Use of such an output function reduces or eliminates linear artefacts, improving the statistical properties of the generator. Blackman and Vigna suggest the scrambling functions: + (addition), * (multiplication), ++ (sum, rotation, sum) and /// (multiplication, rotation, multiplication), which are cheap to execute in modern processors.

xoroshiro128+ has received particular attention because it is the fastest variant of this family and its predecessor xorshift128+ is used in the JavaScript engines of Chrome\(^4\), Firefox and Safari web browsers. However, the use of addition as a non-linearity is known to leave the least significant bits as a weak linear combination of the state vectors, and in particular that bit 0 is just the XOR of its two input bits. Blackman and Vigna acknowledge this weakness by showing that the linearities are detectable by the\ MatrixRank and LinearComp tests of TestU01 only when the least significant bits of the output are placed in the most significant positions. This is because TestU01’s Crush batteries are designed to test random floating-point numbers in the range \([0, 1]\), conversion of random bits biases the higher bits since the lowest bits will be most affected by numerical errors. Blackman and Vigna note

---

2. See the PractRand documentation for details of these tests http://pracrand.sourceforge.net/Tests_engines.txt

3. This relationship is mentioned in the source code for the HWD test: “the Hamming-weight dependency test based on \( z^9 \) from gjrand 4.2.1.0”, http://xoshiro.di.unimi.it/hwd.c.

4. See this 2015 blog post from the Google V8 project: https://v8.dev/blog/math-random.
that MatrixRank and LinearComp can be configured to
detect the linearity without permuting the input bits, and
their own Hamming Weight Dependency test shows up the
linearity after 5 TB of output.

In our own testing, which is detailed in later sections,
we observe the weakness of the lower bits when providing
the byte-reversed 32-bit output to TestU01. Similar results
have been reported by Lemire and O’Neill across various
LFSRs using addition as an output non linearity [14]. More
dramatically, xoroshiro128+ fails the PractRand test suite
after just 128 MB of output on the binary rank test BRank,
and eventually the DC6-5x4Byte-1 test after 32 TB of
output.

The Graphcore IPU’s xoroshiro128aox generator uses the
xoroshiro128 LFSR with a non-linearity operation based
on a sequence of AND, OR and XOR operations. This new
output function has been designed to mitigate the
known linealities in the output, particularly with bit
0, and to be cheap to implement in hardware, particularly
compared with 64-bit addition. From two 64-bit state vectors
\( s_0 = \{s_0_0, s_0_1, \cdots, s_0_{63}\} \) and \( s_1 = \{s_1_0, s_1_1, \cdots, s_1_{63}\} \), result
bit \( i \) of the output \( r \) is defined as:

\[
r_i = s_0_i \oplus s_1_i \oplus ((s_{0_{i-1}} \wedge s_{1_{i-1}}) \vee (s_{0_{i-2}} \wedge s_{1_{i-2}}))
\]

Figure 2 lists a C implementation of the xoroshiro128aox generator, with a function next
that advances the 128-bit state and returns 64 random
bits. This can be compared with the C implementation
of xoroshiro128aox in Figure 2. Note that it uses the
shift constants 55, 14 and 36. Blackman and Vigna later
proposed 24, 16 and 37 as producing superior output. We
include results in this investigation for both variants of the
constants.

4 Generators for comparison

To provide a baseline result, tests are performed against
xoroshiro128+ and with two fast contemporary and com-
parable PRNGs, both with 128 bits of state and 64 bits of
output: philox4x32-10 and pcg64.

Philox [15] is a counter-based family of PRNGs that
have a simple state transition function of an increment
by one, but a complex output function to map state and
key values to pseudorandom outputs. The state transition
makes it easy to jump to arbitrary points in the sequence,
by just setting the counter. We choose the philox4x32-10
variant of this family which has a 128-bit integer counter
and two 32-bit keys as its internal state. Although this
makes the complete state size 192 bits, this is the most
closely comparable version of Philox to xoroshiro128+.
The output of philox4x32-10 is calculated by performing
ten rounds of a scrambling function composed of 32-bit
multiplications and 32-bit XORs. The key values, which are
used as inputs to this are incremented each round by a
constant value. For reference, a C implementation is listed
in Figure 3. The philox4x32-10 generator has established
itself as a standard, and consequently is available as part
of Python scientific computing library NumPy and Nvidia’s
GPU cuRAND library5.

pcg64 is a linear congruential generator (LCG), which
uses multiplication and addition by constants for the state
transition function [16]. To produce outputs, it uses XOR
and rotation operations, in particular using part of the state
vector to set a variable rotation distance. The generator is
specifically characterised by an ‘XSL RR’ output function,
meaning ‘fixed XOR shift to low bits and random rotate’,
and is part of the PCG family of PRNGs that are claimed
to be fast and high quality compared with contemporary
generators. For reference, a C implementation of pcg64 is
listed in Figure 4.

Finally, the 32-bit Mersenne Twister [17] (referred to as
mt32) is included in our analysis since it is the most
widely used PRNG in software, and included as the default
generator in many software systems including Microsoft
Excel, Python and MATLAB. This generator has a state size
of 2.5 KB (624 32-bit words, or 19,937 bits) and a huge period
of 219937 – 1.

The PRNGs used for comparison are tested using ref-
ence C/C++ implementations provided by their authors,
or as part of standard libraries. Additional standard PRNGs
are not included in our analysis due to the computational
cost of performing the statistical tests and because results
for their quality may readily be found in the literature.

5 Methodology

We adopt Vigna’s methodology of sampling generators for
conducting our tests [18]. A generator is tested against a
particular test or test suite suite by choosing 100 seeds
spaced equidistantly in the \( n \)-bit state-space range, that is
at intervals

\[
1 + i \left[ 2^n / 100 \right]
\]

for \( 0 \leq i < 100 \) and obtaining results for all 100 seeds.
In effect, the seeds are chosen randomly with respect to
the sequence produced by a particular generator. It would
be preferable to choose seeds spaced equidistantly in a
generator’s sequence, but it is not always possible for a
generator to jump to arbitrary points, so this method takes
the simplest and most general approach.

For each seed, a generator fails that seed if an extreme
\( p \)-value is reported. We choose the range of extreme \( p \)
values to be \([0.001, 0.999]\) across all tests run, which is
the default range used by TestU01 for reporting failures.
A certain number of failures are always expected however:
assuming all tests contributing to the score are independent,
the probability that a true random number generator pro-
duces a \( p \)-value outside of this range is 0.2%, following a
Poisson distribution with mean 32 and standard deviation
5.7. The criteria for distinguishing a failure is more stringent:
a generator fails a test systematically if it fails all seeds on
the same test. Where systematic failures occur, we report the
test that caused the failure. Only the generators that have
a systematic failure are considered to fail the particular test
set.

6 Results

6.1 TestU01’s BigCrush

In total, we use six different permutations of the 64-bit
PRNG output as input to the BigCrush test suite to avoid
```c
uint64_t s0, s1; // State vectors
uint64_t rotl(uint64_t x, int k) {
    return (x << k) | (x >> (64 - k));
}
uint64_t next(void) {
    // Calculate the result, the 'plus' step.
    const uint64_t result = s0 + s1;
    // State update
    uint64_t sx = s0 ˆ s1;
    s0 = rotl(s0, 55) ˆ sx ˆ (sx << 14);
    s1 = rotl(sx, 36);
    return result;
}
```

**Fig. 1: A C implementation of xoroshiro128plus**

```c
uint64_t s0, s1; // State vectors
uint64_t rotl(uint64_t x, int k) {
    return (x << k) | (x >> (64 - k));
}
uint64_t next(void) {
    uint64_t sx = s0 ˆ s1;
    // Calculate the result, the 'AOX' step.
    uint64_t sa = s0 & s1;
    uint64_t res = sx ˆ (rotl(sa, 1) | rotl(sa, 2));
    // State update
    s0 = rotl(s0, 55) ˆ sx ˆ (sx << 14);
    s1 = rotl(sx, 36);
    return res;
}
```

**Fig. 2: A C implementation of xoroshiro128aox**

| Output | Bits output | Description |
|--------|-------------|-------------|
| std32  | [31:0], [63:32] | All 64 bits used |
| rev32  | [0:31], [32:63] | All 64 bits used |
| std32lo| [31:0]      | Upper 32 bits discarded |
| rev32lo| [0:31]      | Upper 32 bits discarded |
| std32hi| [63:32]     | Lower 32 bits discarded |
| rev32hi| [32:63]     | Lower 32 bits discarded |

**Table 1: Summary of the bits provided to TestU01’s BigCrush for each generator**

Biasing certain bits and to expose known failures. Following standard methodology to avoid biasing of the higher-order bits on conversion to floating-point values in the range [0, 1), the standard output of the generator as well as the reverse is taken (referred to as std32 and rev32 respectively). To demonstrate the systematic failures exhibited by xoroshiro128+ and that this same weakness does not exist in other generators, the reversal of just the lowest 32 bits are taken as output (referred to as rev32lo). We remark that this particular manipulation of the output of the generator is not the only way to expose the weak lower bits to cause a systematic failure. Experimentally, we have found that it is possible to do so by permuting the complete 64-bit output firstly by swapping the high and low 16 bits of each 32-bit output and by a particular interleaving of low and high bits over the full 64-bit output. For completeness, std32lo, std32hi and rev32hi output bit permutations are also included, and are summarised in Table 1.

Table 2 provides a summary of BigCrush test failures for each generator. The number of test failures for a particular generator and output is the total number of test failures across all 100 seeds. The mt32 generator exhibits a systematic failure for the LinearComp test across all output permutations. As expected, xoroshiro128+ exhibits a systematic failure for the LinearComp and MatrixRank tests when run with the rev32lo output. The remaining generators do not exhibit any systematic failures and all can be considered to pass BigCrush. The number of failures for generators that do not fail systematically, fall within three standard deviations of the expected value, 32.

**6.2 PractRand**

Table 3 lists results for the PractRand test set. Both variants of xoroshiro128+ fail quickly as expected, with systematic failures on three instances of the Binary Rank test. The Mersenne Twister lasts longer on the Binary Rank tests, but also fails eventually at 256 GB output. The AOX variants show similar results to pcg64 and philox4x32-10, running to the end of the test data.

Note that unlike BigCrush, the number of test failures does not follow a Poisson distribution given because there are high degrees of correlation between tests and therefore the calculated p-values are not entirely independent, given that there are only five basic tests and thousands of permutations of their parameters.

**6.3 Gjrand**

Table 4 lists the results for the Gjrand test set. The number of test failures is the total number of tests across all 100 seeds that exhibit p-values outside of the range [0.001, 0.999]. All variants of xoroshiro128 fail systematically on both versions of the z9 test, which looks for dependencies in the Hamming Weight of successive outputs. The Mersenne Twister fails Binary Rank (binr) systematically.

**6.4 Hamming Weight Dependency test**

The results for the HWD test are listed in Table 5. Each generator is run until it generates a p-value smaller than $10^{-20}$ or outputs 100 TB of data (which equates to approximately one week of run time). Due to the test runtime, results are given for a single 128-bit seed ($s0 = 1, s1 = -1$). We choose this more extreme p-value bound to be consistent with the results published for other generators [13].
__uint128_t counter;
uint64_t reshi, reslo, count;
uint32_t k0, k1;
uint64_t mul32lo(uint64_t a, uint64_t b) {
    return (a * b) & 0xFFFFFFFFULL;
}
uint64_t mul32hi(uint64_t a, uint64_t b) {
    return ((__uint128_t)2549297995355413924ULL)<<64 | 4865540595714422341ULL;
}
uint64_t next(void) {
    if (counter++ % 2) {
        return reshi;
    } else {
        uint64_t r0, r1, l0, l1;
        r0 = counter & 0xFFFFFFFFULL;
        l0 = (counter >> 32) & 0xFFFFFFFFULL;
        r1 = (counter >> 64) & 0xFFFFFFFFULL;
        l1 = (counter >> 96) & 0xFFFFFFFFULL;
        for (size_t i = 0; i < 10; ++i) {
            l1_next = mul32lo(r1, 0xD2511F53ULL);
            r1_next = mul32hi(r0, 0xCD9E8D57ULL) ^ k0 ^ l0;
            l0_next = mul32lo(r0, 0xCD9E8D57ULL);
            r0_next = mul32hi(r1, 0xD2511F53ULL) ^ k1 ^ l1;
            l1 = l1_next;
            l0 = l0_next;
            r1 = r1_next;
            r0 = r0_next;
            k0 += 0xBB67AE85;
            k1 += 0x9E3779B9;
        }
        reshi = (l1 << 32) | (r1 & 0xFFFFFFFFULL);
        reslo = (l0 << 32) | (r0 & 0xFFFFFFFFULL);
        counter++;
        return reslo;
    }
}

Fig. 3: A C implementation of philox4x32-10

__uint128_t state;
__uint128_t MULTIPLIER = ((__uint128_t)2549297995355413924ULL)<<64 | 4865540595714422341ULL;
__uint128_t INCREMENT = ((__uint128_t)6364136223846793005ULL)<<64 | 144269504088963407ULL;
uint64_t rotr64(uint64_t x, int k) { return (x >> k) | (x << (64 - k)); }
uint64_t next(void) {
    uint64_t result = rotr64(uint64_t(state ^ (state >> 64)), state >> 122);
    state = state * MULTIPLIER + INCREMENT;
    return result;
}

Fig. 4: A C implementation of pcg64

In both shift variants of AOX, biases in the output take substantially more output to detect. These results are comparable to the xoshiro256 (54 TB) and xoroshiro1024+ (36 TB), generators with twice and eight times more state than xoroshiro128aox respectively. It is surprising that the original z9 test of Gjrand detects the dependencies with substantially less output.

Although the Mersenne Twister is a linear generator, it is unsurprising that the HWD test does not detect anomalies because it has an enormous 19,937 bits of state and would require a correspondingly huge amount of memory to do so. When reduced to 607 bits of state, the Mersenne Twister
TABLE 2: Summary of BigCrush test failures for each generator

| Generator          | s32 | r32 | s32l | s32h | r32l | r32h | Total | Systematic failures |
|--------------------|-----|-----|------|------|------|------|-------|---------------------|
| mt32               | 236 | 237 | 233  | 238  | 246  | 237  | 1427  | LinearComp          |
| pcg64              | 34  | 30  | 38   | 37   | 38   | 27   | 204   |                     |
| philox4x32-10      | 33  | 32  | 32   | 32   | 32   | 38   | 195   |                     |
| xoroshiro128+-24-16-37 | 33  | 29  | 28   | 40   | 353  | 42   | 525   | LinearComp, MatrixRank |
| xoroshiro128+-55-14-36 | 31  | 36  | 43   | 39   | 335  | 40   | 524   | LinearComp, MatrixRank |
| xoroshiro128aox-24-16-37 | 31  | 32  | 41   | 30   | 44   | 32   | 210   |                     |
| xoroshiro128aox-55-14-36 | 32  | 33  | 44   | 32   | 41   | 31   | 213   |                     |

TABLE 3: Results for the PractRand test set

| Generator          | Total failures | Total tests | Total output | Systematic failures |
|--------------------|----------------|-------------|--------------|---------------------|
| mt32               | 151            | 36900       | 256 GB       | Low16/64|BRank(12):12K(1) |
| pcg64              | 72             | 45500       | 32 TB        |         |
| philox4x32-10      | 52             | 45500       | 32 TB        |         |
| xoroshiro128+-24-16-37 | 401         | 21300       | 256 MB       | Low4/64|BRank(12):768(1), Low1/64|BRank(12):256(2), Low1/64|BRank(12):384(1) |
| xoroshiro128+-55-14-36 | 402         | 21300       | 256 MB       | Low4/64|BRank(12):768(1), Low1/64|BRank(12):256(2), Low1/64|BRank(12):384(1) |
| xoroshiro128aox-24-16-37 | 77           | 45500       | 32 TB        |         |
| xoroshiro128aox-55-14-36 | 66           | 45500       | 32 TB        |         |

TABLE 4: Results for the Gjrand test set

| Generator          | Test failures | Systematic failures |
|--------------------|---------------|---------------------|
| mt32               | 107           | binr -c             |
| pcg64              | 15            |                     |
| philox4x32-10      | 7             |                     |
| xoroshiro128+-24-16-37 | 257         | z9,z9 -t            |
| xoroshiro128+-55-14-36 | 286         | z9,z9 -t            |
| xoroshiro128aox-24-16-37 | 210         | z9,z9 -t            |
| xoroshiro128aox-55-14-36 | 205         | z9,z9 -t            |

TABLE 5: Results for the HWD test

| Generator          | Bytes output when $p = 10^{-20}$ |
|--------------------|----------------------------------|
| mt32               | >100 TB                          |
| pcg64              | >100 TB                          |
| philox4x32-10      | >100 TB                          |
| xoroshiro128+-24-16-37 | 4 TB     |
| xoroshiro128+-55-14-36 | 4 TB     |
| xoroshiro128aox-24-16-37 | 36 TB     |
| xoroshiro128aox-55-14-36 | 45 TB     |

Fails HWD at 37 GB of output [13].

pcg64 and philox4x32-10 do not exhibit any bias up to 100 TB of output, but neither are based on LFSRs.

7 HARDWARE IMPLEMENTATION COST

To assess the cost of the xoroshiro128aox generator in hardware, we compare it against the contemporary generators included in the statistical testing. To do this, we have produced a Verilog implementations of the generators and performed synthesis and physical place and routing using a Synopsys EDA tool. We use Graphcore’s 7 nm cell library and a target clock period of 1 GHz. In the Verilog implementations, each generator computes its state update and output function in a single cycle, reading and writing to and from registers within the block. Reported gate counts only include combinatorial logic associated with these functions. We omit mt32 from this implementation analysis since the hardware cost of its state alone (19,937 bits) is prohibitively expensive.

The hardware implementation costs of the generators is measured by the number of gates required and the logical depth, which is the maximum number of gates of any path through the logic function. The results of the analysis are summarised in Table 6 and are presented for the state update state update and the output function. Figures 5 to 8 show the scaled floorplans for each generator, including the state-transition logic, output function and all state. The xoroshiro128 LFSR can be implemented with three 64-bit XORs. The shift and rotate operations are by constant values and require no logic. The cost of the AOX output function is similar to the state update, and the full variable 64-bit addition is approximately three times as expensive as AOX.

For pcg64, the main hardware components are a 128-bit constant multiplier and a 128-bit constant adder for the state update, which dominates the hardware cost with ~10,000 gates and 27 levels of logic. Note that there is significant scope for optimising the implementation of constant multipliers since partial products will be generated for each set bit in the constant. Assuming approximately 50% set bits in the constant, only $n/2$ partial products need to be accumulated. The output function requires a 64-bit XOR of the state vector and 64-bit full barrel rotator, which is cheap to implement compared with a full 64-bit adder.

philox4x32-10 is the most expensive generator to implement due to its complex output function consisting of 10 stages of four 32-bit constant multipliers, two 32-bit constant adders and four 32-bit XORs of the $k$ and $l$ values. The implementation cost is ~30,000 cells and 89 stages of
logic. In any practical implementation, the output function would need to be heavily pipelined to meet an acceptable clock speed. Fewer rounds could also be implemented if the quality of the output were satisfactory, however we have not investigated that trade off. The state transition as expected is relatively cheaper, requiring a 128-bit increment by one.

8 Other Aspects of Quality

Apart from statistical quality of the output of a PRNG, several other issues affect the suitability for use in artificial intelligence applications. This section discusses these: period, output uniformity, seeding and overlapping parallel sequences.

8.1 Period

The period of a PRNG is the number of states that are visited before the sequence of states repeats. Since xoroshiro128aox uses the xoroshiro128 LFSR, it too has a period of $2^{128} - 1$, excluding the all-zeros state, from which is cannot transition. This period is sufficient to accommodate many parallel generators, as is discussed in Section 8.4. To give an indication of how large this period is: if a single generator were to output a value every nanosecond (1 billion times a second), it would take $10^{32}$ years to traverse the whole sequence.

8.2 Uniformity

Uniformity of a PRNG is how well distributed the different output values are. A perfectly uniform generator will output all distinct values an equal number of times after completing a full period. A non-uniform generator biases particular values, which is undesirable for RNGs. Since the xoroshiro128 LFSR is full period (notwithstanding the all-zeros state), the analysis of uniformity can be focused on the AOX output function.

We use the $\chi^2$ test for a discrete uniform distribution as a goodness-of-fit test when compared with the $\chi^2$ distribution. Since it is intractable to measure all or even a large part of the full 128-bit state space, the $\chi^2$ statistic is calculated for smaller states and all possible output values, and extrapolated for the 128-bit state size. AOX is a function that maps $2n$-bit state values to $n$-bit outputs. If we take $m$ samples of AOX, then the expected number of occurrences for any output value is $\frac{m}{2^n}$, according to the null hypothesis. Table 7 lists the calculated test statistics for varying state sizes and all state values. In the right-hand column is the critical value for $2^n - 1$ degrees of freedom and a significance level of 95%. The results show that there is no statistically significant difference between the output of AOX and the uniform distribution for any sample sizes, and therefore that the non-uniformity cannot be detectable through sampling of a portion of the 128-bit state space.

8.3 Escaping zero land

A desirable property of a PRNG is that given a 'bad' state where only a minority of bits are set to one, it can rapidly transition to a 'good' state where approximately half the bits are set, such as from a poor initial seed, or a bad state it encountered in its sequence. This capability is often referred to as escaping zero land, and for linear generators is equivalent to the ability for correlated states to decorrelate quickly.

In general, zero escape and decorrelation is a problem for generators with a large state space, where the transition function must spend more time perturbing the state, at the cost of performance/implementation cost.

This issue is relevant to PRNG initialisation since seed values are typically not uniform random bits. In intelligence applications, deterministic execution is important for debugability and so fixed seed values are required. Efficient generation of fixed seeds is important to avoid memory use and to minimise initialisation overheads, but this makes it difficult to ensure they are good values. A similar issue arises when a generator encounters a bad state in its sequence of transitions. In this case, it should recover quickly back to well-balanced states.

To characterise the rate at which a generator escapes from zero land, we use the method of Panneton, L’Ecuyer and Matsumoto [19]. A generator is initialised with a one-hot seed, and the proportion of set bits in the output is recorded over a fixed number of generated values, averaged over the last four outputs. The escape time is calculated by averaging the proportion at each output over all one-hot seeds. The results of this analysis are shown in Figure 9 (1,000 iterations) and Figure 10 (1 million iterations, sampled at intervals of 1,000).

pcg64 and philox4x32-10 produce balanced outputs after four iterations. For philox4x32-10 it is necessary for the output to be balanced regardless of the state since it is advanced as a counter. The AOX output scrambler has a very similar behaviour to addition, with escape time being approximately 14 iterations, relating mainly to the ability of the xoroshiro128 LFSR transition function to decorrelate. The mt32 generator takes over a million cycles to reach an approximately balanced output state, due to it having a much larger state. It should also be noted that the mt32 state does not begin one hot: the Boost C++ implementation used in these experiments performs an operation on the state after it is set directly to improve the balance of set bits. Without this operation, the warmup period would be further extended.

8.4 Overlapping sequences

Since the IPU’s xoroshiro128aox generator is used in the context large amounts of parallelism, with each chip containing more than a thousand processing tiles and a machine containing many chips, it is interesting to ask whether 128 bits of state is sufficient: what is the likelihood that different generators will produce overlapping, and therefore correlated sequences of numbers. Correlation between generators may negatively impact the application, dependent on the way in which the randomness are used, so it is best that such probability is negligible.

To analyse this probability, we can use an upper bound on the probability that a number of sequences of the same length and starting from random points overlap [20]. If $n$ is the number of generators, $L$ is the sequence length and $P$ is the period length, then the probability of overlap is at most $n^2 L / P$. This bound assumes the generator is full
TABLE 6: Hardware costs of different PRNGs as measured by gate counts

| Generator     | State update Total cells | Logic depth | Output function Total cells | Logic depth | Total cells |
|---------------|--------------------------|-------------|----------------------------|-------------|-------------|
| xoroshiro128aox | 331                      | 4           | 353                        | 4           | 684         |
| xoroshiro128plus | 331                     | 3           | 906                        | 13          | 1,237       |
| pcg64         | 9,564                    | 26          | 658                        | 7           | 10,222      |
| philox4x32-10 | 1,003                    | 13          | 29,553                     | 89          | 30,556      |

Fig. 5: xoroshiro128aox floorplan

Fig. 6: xoroshiro128plus floorplan

Fig. 7: pcg64 floorplan

Fig. 8: philox4x32-10 floorplan

Fig. 9: Convergence to half of the output bits being set (1K samples)

Fig. 10: Convergence to half of the output bits being set (1K-1m samples)

period in that there is a single cyclic sequence of transitions between states, which is true for the xoroshiro128 LFSR when excluding the zero state.

Figure 11 shows the probabilities of overlapping sequences for machines with 16, 1,024 and 65,536 IPU chips with a program run time of 1 to 32 days. The following worst-case assumptions are made: 1,216 tiles per chip, 6 contexts per tile, 1 GHz clock speed with a context running at 1/6 GHz and 2 LFSR updates per context cycle, continuously. What this shows is that in an extreme case, with a machine containing 65,536 IPU chips (0.5 bn parallel generators) running for 32 days, the probability of two sequences overlapping is negligible at 0.00006%.

It remains possible however to guarantee that xoroshiro128aox sequences will never overlap by making use of the xoroshiro128 jump functions [21] to move to particular points in the LFSR sequence. These jump functions rely on computed constants for specific jump distances, which may be precomputed in memory, or computed on the fly, particularly if hardware support can be provided for a 128-bit Galois Field multiplier.
that the weakness of the addition output function have been mitigated by AOX. However, LFSR-based PRNGs are known to exhibit linear artefacts that can always be detected given analysis of enough output, or by a particular test. A scrambling of the LFSR’s output can only serve to hide the linearities to some extent. As such, the z9 test of Gjrand and the related HWD test both detect dependencies in the populations of set bits between consecutive outputs for addition and AOX xoroshiro128 variants. Gjrand’s z9 test is most sensitive to this at around 5 TB to 10 TB of output, and HWD in excess of 30 TB. No such dependencies are detected by similar tests in BigCrush or PractRand.

xoroshiro128aox represents a significant improvement in statistical quality over xoroshiro128+ passing both major test suites BigCrush and PractRand, while being cheaper to implement in hardware. Contemporary fast PRNGs with the same state size, pcg64 and philox4x32-10, are more robust to tests for linear artefacts, but are orders of magnitude more expensive to implement in hardware and thus prohibitively expensive for use in the IPU’s tile processors. xoroshiro128aox therefore provides an excellent trade off between implementation cost and statistical quality.

### 9 Conclusions

In this report we have provided a rigorous assessment of the quality of the IPU’s PRNG algorithm, the AOX variant of xoroshiro128. Our analysis goes well beyond the typical testing of PRNGs found in the literature, which often only present results for a particular test suite and sometimes only for a single seed. We provide results for all well-regarded test suites that we are aware of, as well as adopting the approach of sampling generators to test over 100 different sizes of machine.

![Fig. 11: Probability of overlapping sequences with different sizes of machine.](image)

### References

[1] D. Saad, “Online algorithms and stochastic approximations,” *Online Learning*, vol. 5, pp. 6–3, 1998.

[2] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.

[3] M. D. Zeiler and R. Fergus, “Stochastic pooling for regularization of deep convolutional neural networks: 1st international conference on learning representations, iclr 2013,” in *1st International Conference on Learning Representations*, ICLR 2013, 2013.

[4] F. Gressmann, Z. Eaton-Rosen, and C. Luschi, “Improving neural network training in low dimensional random bases,” *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[5] M. Varchola and M. Drutarovsky, “New high entropy element for fpga based true random number generators,” in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2010, pp. 351–365.

[6] S. Vigna, “It is high time we let go of the mersenne twister,” *arXiv preprint arXiv:1910.06437*, 2019.

[7] S. Knowles, “Graphcore,” in *Hot Chips 33 Symposium (HCS)*. IEEE, 2021, pp. 1–25.

[8] “AI-float™ - Mixed precision arithmetic for AI: A hardware perspective,” Graphcore Ltd, Tech. Rep., 2021.

[9] P. L’Ecuyer and R. Simard, “Testu01: A software library in ansi c for empirical testing of random number generators,” http://simul.iro.umontreal.ca/testu01/guideshorttestu01.pdf, 2013, accessed: 2021-08-11.

[10] P. L’Ecuyer and R. Simard, “Testu01: A c library for empirical testing of random number generators,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 33, no. 4, pp. 1–40, 2007.

[11] C. Doty-Humphrey, “Practrand (practically random), a library of pseudo-random number generators and statistical tests,” http://practrand.sourceforge.net, 2021, accessed: 2021-08-11.

[12] D. Blackman, “Gjrand, a library of pseudo-random number generators and statistical testing programs,” http://gjrand.sourceforge.net, 2021, accessed: 2021-08-11.

[13] D. Blackman and S. Vigna, “Scrambled linear pseudorandom number generators,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 47, no. 4, pp. 1–32, 2021.

[14] D. Lemire and M. E. O’Neill, “Xorshift1024*, xorshift1024+, xorshift128+ and xoroshiro128+ fail statistical tests for linearity,” *Journal of Computational and Applied Mathematics*, vol. 350, pp. 139–142, 2019.
J. K. Salmon, M. A. Moraes, R. O. Dror, and D. E. Shaw, “Parallel random numbers: as easy as 1, 2, 3,” in Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12.

M. E. O’Neill, “PCG: A family of simple fast space-efficient statistically good algorithms for random number generation,” ACM Transactions on Mathematical Software, 2014.

M. Matsumoto and T. Nishimura, “Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator,” ACM Transactions on Modeling and Computer Simulation (TOMACS), vol. 8, no. 1, pp. 3–30, 1998.

S. Vigna, “An experimental exploration of marsaglia’s xorshift generators, scrambled,” ACM Transactions on Mathematical Software (TOMS), vol. 42, no. 4, pp. 1–23, 2016.

M. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L’Ecuyer, “Efficient jump ahead for f2-linear random number generators,” INFORMS Journal on Computing, vol. 20, no. 3, pp. 385–390, 2008.

James Hanlon received an M.Eng. degree from the University of Bristol, UK in 2009, and a Ph.D. in computer architecture from the same institution in 2013. Investigating the design of highly parallel machines after this, James worked at a technical consultant for a major UK government agency working on machine benchmarking and technology studies. In 2014, James joined the software tools team in XMOS (Bristol) to upgrade the LLVM compiler to support their second-generation architecture. From 2015, James was involved with early work on a new compiler and toolchain for the IPU processor, prior to Graphcore spinning out as a separate company. In 2016, James moved to the Silicon Engineering team at Graphcore in Bristol. He currently leads the design and implementation of many aspects of the IPU processor, and continues to be interested in the interface between hardware and software.

Diya Rajan received an M.Eng. degree in Electrical and Information Sciences from the University of Cambridge in 2019. She currently works in the Silicon engineering team at Graphcore in Bristol, UK. Diya’s interests include optimising small precision formats for machine learning workloads.

Stephen Felix (M’97) received a B.Eng. in electronic and communication engineering from the University of Bath, UK in 1990. He then joined ST-Microelectronics (formerly Inmos) in Bristol (UK) to design circuits for full-custom microprocessors. In 1994 he went to ST-Catania, Italy to develop a processor version for set-top-boxes. In 1997, Stephen joined the Alpha Microprocessor design team in Massachusetts where he designed the micro-architecture and circuits for an 8-way superscalar SMT processor. In 2001 he joined Intel (MA) to develop a multi-core microprocessor and ring-interconnected, shared L3 cache. In 2002 Stephen joined Icera Inc. (UK) at its inception where he co-developed an ISA and custom microprocessor for high-performance software-based cellular modems. Since 2015, he has been with Graphcore Ltd in Bristol and is an Engineering Fellow. Current interests include all aspects of chip, hardware and systems design and technology for high-performance machine learning.