Fast and Small Nonlinear Pseudorandom Number Generators for Computer Simulation

Samuel Neves and Filipe Araujo

CISUC, Dept. of Informatics Engineering
University of Coimbra, Portugal
{sneves,filipius}@dei.uc.pt

Abstract. In this paper we present Tyche, a nonlinear pseudorandom number generator designed for computer simulation. Tyche has a small 128-bit state and an expected period length of 2^{127}. Unlike most nonlinear generators, Tyche is consistently fast across architectures, due to its very simple iteration function derived from ChaCha, one of today's fastest stream ciphers.
Tyche is especially amenable for the highly parallel environments we find today, in particular for Graphics Processing Units (GPUs), where it enables a very large number of uncorrelated parallel streams running independently. For example, 2^{16} parallel independent streams are expected to generate about 2^{96} pseudorandom numbers each, without overlaps.
Additionally, we determine bounds for the period length and parallelism of our generators, and evaluate their statistical quality and performance. We compare Tyche and the variant Tyche-i to the XORWOW and TEA_8 generators in CPUs and GPUs. Our comparisons show that Tyche and Tyche-i simultaneously achieve high performance and excellent statistical properties, particularly when compared to other nonlinear generators.

Keywords: ChaCha, GPU, PRNG, random number generation, SIMD, Tyche, Tyche-i

1 Introduction

Pseudorandom numbers are often used for testing, simulation and even aesthetic purposes. They are an integral part of Monte Carlo methods, genetic and evolutionary algorithms, and are extensively used in noise generation for computer graphics.

Monte Carlo methods were first used for computing purposes by Ulam and von Neumann, while attempting to solve hard problems in particle physics [1]. Monte Carlo methods consist of iterated random sampling of many inputs in some probability distribution, followed by later processing. Given enough inputs, it is possible to obtain an approximate solution with a reasonable degree of certainty. This is particularly useful for problems with many degrees of freedom, where analytical or exact methods would be far too inefficient. Monte Carlo methods are not, however, very computationally efficient — typically, to reduce the error margin by half, one has to quadruple the amount of sampled inputs [2]. Today, Monte Carlo methods are used in the most various fields, such as particle physics, weather forecasting, financial analysis, operations research, etc.

Current general-purpose processors typically have 2 or 4 cores. Graphics processing units have tens to hundreds [3]; future architectures are slated to scale up to hundreds and thousands of cores [4]. This development entails some consequences: silicon real estate is limited, and the increase in processing units decreases the total fast memory available on-chip. Thus, it becomes an interesting problem to design a random number generator that can take advantage of massively parallel architectures and still remain small, fast and of high quality. With these goals in mind, we introduce Tyche, a fast and small-state pseudorandom number generator especially suited for current parallel architectures. The iteration function of Tyche, derived from the stream cipher ChaCha's *quarter-round* function [5], is nonlinear and fast; it uses a very small amount of state (4 32-bit registers) and, yet, it has a very large average period.

In Section 2 we review the state of the art in theory and practice of random number generation. In Section 3 we describe the Tyche function. In Section 4 we provide an analysis of several important features of the algorithm, such as the expected period, statistical quality and parallelism. We then introduce a variant of Tyche with higher instruction-level parallelism in Section 5. In Section 6, we experimentally evaluate and compare Tyche. Section 7 concludes the paper.

2 Related Work

There is an enormous body of work in the literature regarding pseudorandom number generation. One of the first and most popular methods to generate pseudorandom numbers in digital computers was Lehmer's linear congruential method, consisting of a linear recurrence modulo some integer m. Since then, researchers have proposed numerous other linear algorithms, most notably lagged Fibonacci generators, linear feedback shift registers, Xorshift generators and the Mersenne Twister [6,7,8]. The statistical properties of linear generators are well known and widely studied; several empirical tests are described in [6, Chapter 3]. One of the drawbacks of such linear generators, in particular linear congruential generators, is their very regular lattice structure [6, Section 3.3.4]. This causes the usable amount of numbers in a simulation to be far less than the full period of the generator [9].

Nonlinear pseudorandom generators, like the *Inversive congruential generator* and *Blum Blum Shub* [10,11], avoid the drawbacks of linearity. However, nonlinear generators often require more complex arithmetic than linear ones and have smaller periods, rendering them impractical for simulations.

The generation of pseudorandom numbers in parallel environments is also a well studied subject [12,13,14]. The main problem is to enable multiple concurrent threads of execution to get random numbers in parallel. Two main solutions exist for this problem: *parametrization* and *cycle splitting*. Parametrization consists in creating a slightly different full period generator for each instance; this can be done by, *e.g.*, changing the iteration function itself. Cycle splitting takes a full period sequence and splits it into a number of subsequences, each used within an instance. Cycle splitting is often used in linear congruential generators, since it is possible to leap to any arbitrary position in the stream quite easily. Other generators, such as the Mersenne Twister, rely on different initial parameters (*i.e.*, parametrization) to differentiate between threads.

In the case of GPUs and other vector processors, we face additional restrictions, because the amount of fast memory per core is quite limited, thus restricting the internal state we can use. Furthermore, GPUs lack some important instructions, such as native integer multiplication and/or division, leading to a large slowdown for some popular random number generators. Still, linear congruential generators have been studied in GPUs [15].

There have been some initial attempts to adapt cryptographic functions for fast GPU random number generation. Tzeng and Wei [16] used the MD5 hash function and reduced-round versions thereof to generate high-quality random numbers. Zafar and Olano [17] used the smaller and faster 8-round TEA block cipher to generate high-quality random numbers for Perlin noise generation.

3 Tyche

This section will present Tyche. In the following sections, all values are assumed to be 32 bits long, unless otherwise noted. $+$ represents addition modulo 2^{32}; $\oplus$ denotes the exclusive-or (xor) operation; $\lll$ means bitwise rotation towards the most significant bits.

3.1 Initialization

The state of Tyche is composed of 4 32-bit words, which we will call a, b, c and d. Tyche, when initialized, takes a 64-bit integer *seed* and a 32-bit integer *idx*. Algorithm 3.1 describes the operations performed during initialization.

Algorithm 3.1: TYCHE INIT$(a, b, c, d, seed, idx)$

```
$a \leftarrow \lfloor seed/2^{32} \rfloor$
$b \leftarrow seed \bmod 2^{32}$
$c \leftarrow 2654435769$
$d \leftarrow 1367130551 \oplus idx$
for $i \leftarrow 0$ to $20$
  do $MIX(a, b, c, d)$
return $(a, b, c, d)$
```

The MIX function called in Algorithm 3.1 is used here to derive the initial state; it is described further in Section 3.3. The constants used in the initialization, 2654435769 and 1367130551, were chosen to be $\lfloor 2^{32}/\varphi \rfloor$ and $\lfloor 2^{32}/\pi \rfloor$, where φ is the golden ratio and π is the well-known constant. Their purpose is to prevent a starting internal state of $(0, 0, 0, 0)$.

3.2 The algorithm

Once its internal state is initialized, Tyche is quite simple. It calls the MIX function once and returns the second word of the internal state, as shown in Algorithm 3.2.

Algorithm 3.2: $\textsc{Tyche}(a, b, c, d)$

```
$(a, b, c, d) = MIX(a, b, c, d)$
**return** $(b)$
```

3.3 The MIX function

The MIX function, used both in initialization and state update, is derived directly from the *quarter-round* function of the ChaCha stream cipher [5]. As described in Algorithm 3.3, it works on 4 32-bit words and uses only addition modulo 2^{32}, XOR and bitwise rotations.

Algorithm 3.3: $\mathrm{MIX}(a, b, c, d)$

```
$a \leftarrow a + b$
$d \leftarrow (d \oplus a) \lll 16$
$c \leftarrow c + d$
$b \leftarrow (b \oplus c) \lll 12$
$a \leftarrow a + b$
$d \leftarrow (d \oplus a) \lll 8$
$c \leftarrow c + d$
$b \leftarrow (b \oplus c) \lll 7$
**return** $(a, b, c, d)$
```

4 Analysis of Tyche

4.1 Design

We can find many different designs for random number generators. The design we propose here attempts to achieve high period, speed, and very low memory consumption. One of the ways in which it achieves this is by using a very simple recursion:

$$x_{i+1} = f(x_i) \qquad (1)$$

This requires no extra space other than the state's size and perhaps some overhead to execute f. One could use, *e.g.*, a counter to ensure certain minimum period — this would evidently require more registers per state, which goes against our main objectives. A similar approach to ours has been used in the

LEX [18] stream cipher, using the AES block cipher in Output Feedback mode (OFB) and extracting 4 bytes of the state per iteration.

Another crucial design choice concerns function f. Should it be linear? Most current random number generators are indeed linear: LCG, Xorshift, LFSR constructions, etc. These functions have the advantage of being very simple and easily analyzed. However, linear random number generators tend to have highly regular outputs: their outputs lie on simple lattice structures of some dimension. This makes such generators unsuitable for some types of simulations and considerably reduces their useful period [19]. Nonlinear generators generally do not have this problem [2]. Moreover, despite being very simple, linear generators may not be very fast. Linear congruential generators and their derivatives require multiplications and modular reductions. Unfortunately, these operations are not present in every instruction set and can be hard to implement otherwise.

One could then simply search for a good nonlinear random number generator. However, nonlinear generators for simulation purposes are hard to find, and generally much slower than their linear counterparts. Indeed, one could simply use a cryptographic stream cipher as a random number generator. That would, however, be several times slower and would require a much larger state. Indeed, even TEA_8 as described in [17] requires 136 instructions per 64 bits of random output, while MIX only requires 12 instructions per 32 bits of random output.

In light of these reasons, we chose our function to be nonlinear and to use exclusively instructions available in almost every chip — addition, xor, bit rotations[1]. The overlap of 32-bit addition and xor creates a high amount of nonlinearity and simultaneously allows for very fast implementations, owing to the simplicity of such instructions.

4.2 Period

The MIX function, used to update the internal state, is trivially reversible. Thus it is a permutation, with only one possible state before each other state. How does this affect the expected period? Were the MIX function irreversible, it would behave like a random mapping—in that case, the period would be about $2^{n/2}$ for an n-bit state [20]. In our case, the expected period is the average cycle length of a random element in a random permutation: $(2^n + 1)/2 \approx 2^{n-1}$ for an n-bit state [21, Section 1.3.3].

It is also known that random permutations do have small-length cycles. In fact, we can trivially find one cycle of length 1 in the MIX function: MIX(0,0,0,0) = (0,0,0,0) — this is in fact its only fixed point [22]. However, if using the initialization described in Section 3.1, this state will never be reached. It is also extremely unlikely to reach a very short cycle—the probability of reaching a cycle of length m is $1/2^n$; the probability of reaching a cycle of length *m or less* is $\sum_i^m 1/2^n = m/2^n$ [23]. In our case, the chance of reaching a state with period less than or equal to 2^{32} is roughly 2^{-96}.

[1] While many chips do not have bit rotations natively, they are trivially achievable with simple logical instructions such as shifts and xor.

4.3 Parallelization

Our algorithm is trivial to use in parallel environments. When initializing a state (using Algorithm 3.1 or 5.1), each computing unit (*e.g.*, thread, vector element, core) uses the same 64-bit seed, but its own index in the computation (the `idx` argument of Algorithm 3.1). We chose a 64-bit seed to avoid collisions; since seeds are often chosen at random, it would only require about 2^{16} initializations for a better than 50% chance to rerun a simulation using the same seed if one used 32-bit seeds. This would be unacceptable.

What about overlaps? Parallel streams will surely overlap eventually, given that the function is cyclic and reversible. This is as bad as a small period in a random number generator. To find out how fast streams overlap, consider a simple case: s streams outputting a single value each. Given that each stream begins at an arbitrary state out of n possible states, the probability of an overlap (*i.e.*, a collision) would be given by the birthday paradox:

$$1 - \frac{n!}{(n-s)!n^s} \tag{2}$$

This is, however, a simplified example; what we want to know is the likelihood that, given s streams and a function f that is a random permutation, no stream will meet the starting point of any other in less than d calls to f. This can be seen as a generalization of the birthday problem, and was first solved by Naus [24]. The probability that at least one out of s streams overlaps in less than d steps in a cycle of length m is given by

$$1 - \frac{(m-sd+s-1)!}{(m-sd)!m^{s-1}} \tag{3}$$

In our particular case, m is in average 2^{127}; s should be no more than 2^{16}; d should be a large enough distance to make the generator useful — we choose 2^{64} here as an example minimum requirement. Thus, 2^{16} parallel streams producing 2^{64} numbers will overlap with a probability of roughly 2^{-32}. Conversely, when running 2^{16} parallel streams, an overlap is not expected until about $2^{64}/2^{-32} = 2^{96}$ iterations have passed.

5 A Faster Variant

One issue with the construction described in the previous section is that it is completely sequential. Each instruction of the MIX function directly depends on the immediately preceding one. This does not take any advantage of modern superscalar CPU machinery. Thus, we propose a variant of Tyche, which we call

Tyche-i, able to take advantage of pipelined processors. Tyche-i is presented in Algorithms 5.1, 5.2, and 5.3.

Algorithm 5.1: TYCHE-I INIT$(a, b, c, d, seed, idx)$

```
$a \leftarrow \lfloor seed/2^{32} \rfloor$
$b \leftarrow seed \bmod 2^{32}$
$c \leftarrow 2654435769$
$d \leftarrow 1367130551 \oplus idx$
**for** $i \leftarrow 0$ **to** $20$
  **do** MIX-i$(a, b, c, d)$
**return** $(a, b, c, d)$
```

Algorithm 5.2: TYCHE-I(a, b, c, d)

```
$(a, b, c, d) = \text{MIX-i}(a, b, c, d)$
**return** $(a)$
```

Algorithm 5.3: MIX-I(a, b, c, d)

```
$b \leftarrow (b \ggg 7) \oplus c; c \leftarrow c - d$
$d \leftarrow (d \ggg 8) \oplus a; a \leftarrow a - b$
$b \leftarrow (b \ggg 12) \oplus c; c \leftarrow c - d$
$d \leftarrow (d \ggg 16) \oplus a; a \leftarrow a - b$
**return** $(a, b, c, d)$
```

The main difference between Tyche and Tyche-i is the MIX-i function. The MIX-i function is simply the *inverse function* of Tyche's MIX. Unlike MIX, MIX-i allows for 2 simultaneous executing operations at any given time, which is a better suit to superscalar processors than MIX is. The downside, however, is that MIX-i diffuses bits slower than MIX does: for 1-bit differences in the internal state, 1 MIX call averages 26 bit flipped bits, while MIX-i averages 8.

6 Experimental Evaluation

6.1 Performance

We implemented and compared the performance of Tyche and Tyche-i against the XORWOW generator found in the CURAND library [25]. XORWOW is a combination of a Xorshift [8] and an additive generator, with a combined period of $2^{192} - 2^{32}$. The test setup was the Intel Core 2 E8400 processor for the CPU benchmarks, and the NVIDIA GTX580 GPU for the GPU benchmarks. Table 1 summarizes our performance results in both architectures; note that GPU figures do not take into account kernel and memory transfer overheads, as those are essentially equal for every option.

Table 1: Period, state size, results of TestU01's "BigCrush", and performances, in cycles per 32-bit word, of various pseudorandom number generators in the CPU and GPU.

Algorithm	Period	State	BigCrush	CPU	GPU
Tyche	$\approx 2^{127}$	128	160/160	12.327321	1.156616
Tyche-i	$\approx 2^{127}$	128	160/160	6.073590	0.763572
XORWOW [25]	$2^{192} - 2^{32}$	192	157/160	7.095927	0.578620
TEA_8 [26]	2^{64}	64	160/160	49.119271	5.641948

As expected (cf. Section 5), Tyche-i is roughly twice as fast as Tyche on the Core 2, a processor with high instruction-level parallelism. In the GPU, Tyche-i is still quite faster, but by a lower (roughly 1.5) ratio.

The XORWOW algorithm only requires bit shifts, not bit rotations. Unfortunately, the Fermi GPU architecture does not support native bit rotations, which are replaced by two shifts and a logical operation. This explains the slight speed advantage of XORWOW. Note that we are able to improve 2 out of the 4 rotations by using the Fermi `PRMT` instruction, which allows one to permute bytes of a word arbitrarily. On the CPU, native rotations are as fast as shifts, and Tyche-i actually beats XORWOW in speed.

We also include a comparison with TEA_8, which reveals to be markedly slower than any of the other choices for GPU (and CPU) random generation. As we already pointed out in Section 4.1, TEA_8 requires at least 136 instructions per 64-bit word, which is much higher than either Tyche, Tyche-i or XORWOW.

6.2 Statistical quality tests

In order to assess the statistical quality of Tyche and Tyche-i, we performed a rather exhaustive battery of tests. We employed the ENT and DIEHARD suites and the TestU01 "BigCrush" battery of tests [27,28,29]. Every test performed in both variants showed no statistical weaknesses (cf. Table 1).

Another aspect of Tyche is that it is based on the ChaCha stream cipher. ChaCha's "quarter-round" function is also employed, albeit slightly modified, in the BLAKE SHA-3 candidate[22]. The "quarter-round" has been extensively analyzed for flaws, but both functions are still regarded as secure [30,31]. This increases our confidence in the quality of Tyche as a generator.

Finally, note that the XORWOW algorithm fails 3 tests in the "BigCrush" battery: CollisionOver ($t = 7$), SimpPoker ($r = 27$), and LinearComp ($r = 29$), the latter being a testament of its linear nature.

7 Conclusion

In this paper we presented and analyzed Tyche and Tyche-i, fast and small nonlinear pseudorandom generators based on the ChaCha stream cipher building blocks.

Tyche and Tyche-i use a very small amount of state that fits entirely into 4 32-bit registers. Our experiments show that Tyche and Tyche-i are much faster than the also nonlinear and cryptographic function-derived TEA$_8$, while exhibiting a large enough period for serious simulations with many parallel threads. On the other hand, when we compare Tyche and Tyche-i to the slightly faster (but linear) XORWOW algorithm, statistical tests (*i.e.*, BigCrush) suggest that both Tyche and Tyche-i have better statistical properties.

Acknowledgments

This work has been supported by the project CMU-PT/RNQ/0015/2009, TRONE — Trustworthy and Resilient Operations in a Network Environment.

References

1. Metropolis, N., Ulam, S.: The Monte Carlo Method. Journal of the American Statistical Association **44**(247) (1949) 335–341
2. Gentle, J.E.: Random Number Generation and Monte Carlo Methods. Second edition edn. Springer-Verlag (2003)
3. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture. IEEE Micro **28**(2) (2008) 39–55
4. Vangal, S.R., Howard, J., Ruhl, G., Dighe, S., Wilson, H., Tschanz, J., Finan, D., Singh, A., Jacob, T., Jain, S., Erraguntla, V., Roberts, C., Hoskote, Y., Borkar, N., Borkar, S.: An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. Solid-State Circuits, IEEE Journal of **43**(1) (2008) 29–41
5. Bernstein, D.J.: ChaCha, a variant of Salsa20. `http://cr.yp.to/papers.html#chacha` (January 2008)
6. Knuth, D.E.: Art of Computer Programming. Volume 2: Seminumerical Algorithms. 3 edn. Addison-Wesley Professional (November 1997)
7. Matsumoto, M., Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Trans. Model. Comput. Simul. **8**(1) (1998) 3–30
8. Marsaglia, G.: Xorshift RNGs. Journal of Statistical Software **8**(14) (07 2003)
9. Pawlikowski, K., Jeong, H.D., Lee, J.S.R.: On Credibility of Simulation Studies of Telecommunication Networks. IEEE Communications Magazine (January 2002) 132–139
10. Hellekalek, P.: Inversive Pseudorandom Number Generators: Concepts, Results, and Links. In Alexopoulos, C., Kang, K., Lilegdon, W.R., Goldsman, D., eds.: Proceedings of the 1995 Winter Simulation Conference, IEEE Press (1995) 255–262
11. Blum, L., Blum, M., Shub, M.: A Simple Unpredictable Pseudo-Random Number Generator. SIAM J. Comput. **15**(2) (1986) 364–383
12. Eddy, W.F.: Random Number Generators for Parallel Processors. Journal of Computational and Applied Mathematics **31** (1990) 63–71
13. Brent, R.: Uniform random number generators for supercomputers. In: Proc. Fifth Australian Supercomputer Conference, Melbourne (December 1992) 95–104

14. Schoo, M., Pawlikowski, K., McNickle, D.: A Survey and Empirical Comparison of Modern Pseudo-Random Number Generators for Distributed Stochastic Simulations. Technical report, Department of Computer Science and Software Development, University of Canterbury (2005)
15. Langdon, W.B.: A fast high quality pseudo random number generator for nvidia cuda. In: GECCO '09: Proceedings of the 11th annual conference companion on Genetic and evolutionary computation conference, New York, NY, USA, ACM (2009) 2511–2514
16. Tzeng, S., Wei, L.Y.: Parallel white noise generation on a GPU via cryptographic hash. In: Proceedings of the 2008 symposium on Interactive 3D graphics and games. I3D '08, New York, NY, USA, ACM (2008) 79–87
17. Zafar, F., Olano, M., Curtis, A.: GPU random numbers via the tiny encryption algorithm. In: Proceedings of the Conference on High Performance Graphics. HPG '10, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association (2010) 133–141
18. Biryukov, A.: The Design of a Stream Cipher LEX. In Biham, E., Youssef, A.M., eds.: Selected Areas in Cryptography. Volume 4356 of Lecture Notes in Computer Science., Springer (2006) 67–75
19. L'Ecuyer, P., Simard, R.: On the performance of birthday spacings tests with certain families of random number generators. Math. Comput. Simul. **55**(1-3) (2001) 131–137
20. Flajolet, P., Odlyzko, A.M.: Random mapping statistics. In: EUROCRYPT '89: Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology, New York, NY, USA, Springer-Verlag New York, Inc. (1990) 329–354
21. Knuth, D.E.: Art of Computer Programming. Volume 1: Fundamental Algorithms. Addison-Wesley (July 2002)
22. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. Submission to NIST (Round 3) (2010)
23. Chambers, W.G.: On Random Mappings and Random Permutations. In Preneel, B., ed.: FSE. Volume 1008 of Lecture Notes in Computer Science., Springer (1994) 22–28
24. Naus, J.I.: An extension of the birthday problem. The American Statistician **22**(1) (February 1968) 27–29 `http://www.jstor.org/stable/2681879`.
25. NVIDIA: CUDA Toolkit 4.0 CURAND Guide. (January 2011)
26. Zafar, F., Curtis, A., Olano, M.: GPU Random Numbers via the Tiny Encryption Algorithm. In: HPG 2010: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on High Performance Graphics, Saarbrucken, Germany (June 2010)
27. Walker, J.: A Pseudorandom Number Sequence Test Program. `http://www.fourmilab.ch/random/` (January 2008)
28. Marsaglia, G.: The Marsaglia random number CDROM including the DIEHARD battery of tests of randomness. See `http://stat.fsu.edu/pub/diehard` (1996)
29. L'Ecuyer, P., Simard, R.: TestU01: A C library for empirical testing of random number generators. ACM Trans. Math. Softw. **33**(4) (2007) 22
30. Aumasson, J.P., Fischer, S., Khazaei, S., Meier, W., Rechberger, C.: New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba. (2008) 470–488
31. Aumasson, J.P., Guo, J., Knellwolf, S., Matusiewicz, K., Meier, W.: Differential and invertibility properties of BLAKE. In: Proceedings of the 17th international conference on Fast software encryption. FSE'10, Berlin, Heidelberg, Springer-Verlag (2010) 318–332