Parallel Random Numbers: As Easy as 1, 2, 3

John K. Salmon,[*] Mark A. Moraes, Ron O. Dror, and David E. Shaw[*][†]

D. E. Shaw Research, New York, NY 10036, USA

ABSTRACT

Most pseudorandom number generators (PRNGs) scale poorly to massively parallel high-performance computation because they are designed as sequentially dependent state transformations. We demonstrate that independent, keyed transformations of counters produce a large alternative class of PRNGs with excellent statistical properties (long period, no discernable structure or correlation). These *counter-based* PRNGs are ideally suited to modern multi-core CPUs, GPUs, clusters, and special-purpose hardware because they vectorize and parallelize well, and require little or no memory for state. We introduce several counter-based PRNGs: some based on cryptographic standards (AES, Threefish) and some completely new (Philox). All our PRNGs pass rigorous statistical tests (including TestU01's BigCrush) and produce at least 2^{64} unique parallel streams of random numbers, each with period 2^{128} or more. In addition to essentially unlimited parallel scalability, our PRNGs offer excellent single-chip performance: Philox is faster than the CURAND library on a single NVIDIA GPU.

1. INTRODUCTION

Pseudorandom number generators (PRNGs) have been an essential tool in computer simulation, modeling, and statistics since the earliest days of electronic computation [46], so the development of PRNGs that map naturally onto current and future high-performance architectures is critical to a wide variety of applications [19]. In recent years, the exponential growth of the number of transistors on a chip predicted by Moore's law has delivered more parallelism, but constant (or even decreasing) clock speed. The result is an increasing dominance of multi-core architectures, stream-oriented graphics processors (GPUs), streaming SIMD instruction extensions (such as SSE), and special-purpose hardware. Unfortunately, good parallel PRNGs, especially ones that scale effectively to the level

needed by massively parallel high-performance computers, are still hard to find [16, 36].

Much of the difficulty with parallelizing conventional PRNGs arises from the fact that the algorithms on which they are built are inherently sequential. Conventional PRNGs operate by successively applying a transformation function, f, which maps an element, s_n, in a state space, $\mathcal{S}$, to the next element, s_{n+1}, in sequence:

$$s_{n+1} = f(s_n). \tag{1}$$

Important statistical properties of a PRNG follow from the size of $\mathcal{S}$, which thus must be large, and f, which must thus be complicated. More importantly, the procedure is fundamentally serial: each value depends on the previous one.[1]

There are two approaches to parallelizing a PRNG—multistream and substream—but both approaches have practical difficulties for conventional PRNGs. In the multistream approach, the PRNG algorithm is instantiated in parallel with different parameters, so that each instance produces a distinct stream of numbers. This approach is only viable for families of PRNGs that have large, easily enumerable sets of "known good" parameters, which have proven elusive. In the substream approach, a single logical sequence of random numbers is subdivided into disjoint substreams that may be accessed in parallel [6, 13, 15, 28]. The substream approach requires a mechanism for partitioning the much longer underlying sequence. Only a few PRNGs allow the state space to be deterministically partitioned, and there is a danger that the generated streams are not statistically independent [7, 8, 15]. If the state space is large enough, one can partition probabilistically by choosing starting points for the parallel streams "at random." This requires an underlying PRNG with a very long period so that the probability of overlap between substreams is negligible.[2] Initializing the state of long-period PRNGs is nontrivial, and failure to do so properly can result in significant defects [30].

Regardless of which underlying PRNG and approach is chosen, parallel operation of a PRNG requires storage for the state in each parallel computational unit (task, process, thread, warp, core, node, etc.). On-chip memory in close proximity to the ALU is a precious resource on mod-

[*]E-mail correspondence: John.Salmon@DEShawResearch.com and David.Shaw@DEShawResearch.com.

[†]David E. Shaw is also with the Center for Computational Biology and Bioinformatics, Columbia University, New York, NY 10032.

[1]For some conventional PRNGs, it is possible to "leapfrog" over multiple output values in less time than it takes to repeatedly invoke f, but this does not change the fact that the specification and the most common implementations are fundamentally serial.
[2]The probability of overlapping substreams is a "birthday problem" [31].

ern hardware such as GPUs, which makes generators with very large state spaces less desirable. The time and space required to initialize and store the 2.5-kB-per-thread state of the popular Mersenne Twister [29], for example, may cause significant overhead for a highly parallel GPU application or for specialized hardware.

In this paper, we explore an alternative approach to the design of PRNGs and demonstrate its suitability to parallel computation. We originally used this approach in biomolecular simulations on Anton [40, 41], a massively parallel special-purpose machine, but the approach is far more general. Ignoring some details (see Section 2 for a complete definition), the sequence is:

$$x_n = b(n). \qquad (2)$$

The nth random number is obtained directly by some function, b, applied to n. In the simplest case, n is a p-bit integer counter, so we call this class of PRNGs *counter-based*. Equation 2 is inherently parallel; that is, it eliminates the sequential dependence between successive x_n in Eqn 1. The nth element is just as easy to compute as the "next" element, and there is no requirement or incentive to generate random values in any particular order. Furthermore, if b is a "bijection" (a one-to-one mapping of the set of p-bit integers onto itself), then the period of the generator defined by Eqn 2 is 2^p.

It may not be obvious that practical functions, b, exist that generate high-quality random numbers using Eqn 2. Cryptographers, however, have been designing entire families of functions—called block ciphers, pseudo-random permutations, or pseudo-random bijections—for a generation. Such functions have exactly the properties needed for b in Eqn 2. In Section 3, we review some basic techniques of modern cryptography, with an emphasis on aspects that are relevant for random number generation.

Cryptographic block ciphers are generally "keyed" functions, such as

$$x_n = b_k(n), \qquad (3)$$

where k is an element of a key space and b_k is one member of a family of bijections. A counter-based PRNG constructed from a keyed bijection can be easily parallelized using either the multistream approach over the key space, or the substream approach over the counter space.

Counter-based PRNGs are ideally suited to parallel computation because they break the sequential dependence between output values. It is also important to ensure that no sequential dependence is imposed by the software model and application programming interface (API). We discuss a natural API for counter-based PRNGs that gives applications complete control over either multistream or substream generation of random numbers by controlling keys as well as counters. Applications can choose to derive k and n on the fly from either machine parameters (e.g., core number, MPI rank, node number, thread number, etc.) or from application variables (e.g., particle identifier, voxel number, data block identifier, grid index, etc.). Generating random numbers from state associated with application variables allows for machine-independent streams of random numbers (even with different levels of physical parallelism, since the streams are not associated with machine parameters). This approach permits deterministic results across different computing platforms, a boon for debugging and repeatability.

Such use of application variables for random number streams is nearly impossible to achieve with conventional PRNGs because enormous memory resources would be required to store the state of so many (up to billions of) conventional PRNGs.

In Section 4, we report on three families of PRNGs, all of which are *Crush-resistant*, that is, they pass the extensive empirical validation batteries described in Section 2.2 with zero failures.[3] These tests include the complete SmallCrush, Crush, and BigCrush batteries in TestU01 [24] as well as additional tests that are impractical for conventional PRNGs because they rely on random access to the PRNG's output.

All three families have PRNGs with periods of 2^{128} or longer, and are parameterized by a key that allows for 2^{64} or more distinct, parallel streams. All three families have state and key spaces that are frugal with the precious on-chip memory of modern hardware. All of them have a small "granularity" (i.e., the amount of data returned by each "turn of the crank"), thus reducing buffering requirements for an application that needs only a few random values at a time. Performance ranges from competitive to outstanding (see Table 2 and Section 5); one of the PRNGs we introduce is the fastest PRNG we know of on CPUs, and another is the fastest we know of on GPUs.

The first family consists of bona fide cryptographic block ciphers. One of these (AES [34]) has previously been studied in the context of random number generation [17], but has not been widely adopted because its performance is thought to be significantly below that of conventional PRNGs. We reconsider AES as a PRNG in light of the recent availability of of the AES New Instructions (AES-NI) on commodity x86 processors [14]. On hardware with such support, a PRNG based on AES has a 16-byte key size and granularity, requires 176 bytes of state for each keyed multistream, and has performance that exceeds the fastest conventional PRNGs.

AES is an order of magnitude slower without hardware support. Threefish [11] is a cryptographic block cipher with more portable performance characteristics. A PRNG based on Threefish-256 has a granularity of 32 bytes. The key can be as large as 48 bytes, but needs no internal state, and an application is under no obligation to use all (or any) of the key; unused key space need not consume memory. Threefish is significantly faster than software AES, but its speed is still well below that of popular conventional PRNGs.

The second family consists of PRNGs based on reducing the number of rounds and simplifying the key schedule in cryptographic ciphers. We call two examples of this family *ARS* (Advanced Randomization System) and *Threefry* to avoid confusion with their secure ancestors. ARS requires no storage per key, and on CPUs with AES-NI support, it is our fastest Crush-resistant PRNG. On CPUs without hardware AES support, Threefry is the fastest Crush-resistant PRNG, requiring only common bitwise operators and integer addition. Threefry can be expected to perform well across a wide range of instruction set architectures.

The third family consists of non-cryptographic bijections. In this family, we investigate Philox, a new, non-cryptographic bijection that uses multiplication instructions that compute the high and low halves of the

---

[3]This is a formidable challenge. Many of the most widely used PRNGs, including those standardized by C, C++0x and POSIX fail multiple tests.

product of word-sized operands. We consider Philox variants with periods of 2^{128} and 2^{256}. Philox is the fastest Crush-resistant random number generator we know of on GPUs.

2. COUNTER-BASED RANDOM NUMBER GENERATION

Generalizing L'Ecuyer's definition [22], we define a keyed family of PRNGs in terms of an internal state space, $\mathcal{S}$; an output space, $\mathcal{U}$; a key space, $\mathcal{K}$; an integer output multiplicity; J; a state transition function, f; and an output function, g:

$$f : \mathcal{S} \to \mathcal{S} \qquad \text{is the transition function,}$$
$$g : \mathcal{K} \times \mathbb{Z}_J \times \mathcal{S} \to \mathcal{U} \qquad \text{is the output function.}$$

The output function, $g_{k,j}$, has two indices. The first, k, is a member of the key space, $\mathcal{K}$, which allows us to consider distinct streams of random numbers generated by the same algorithm and parameters, but differing by a key. The second index, $j \in \mathbb{Z}_J$, allows us to extract a small number, J, of output values from each internal state of the PRNG. Using $J = 4$, for example, would allow a PRNG with a 128-bit state space to produce four 32-bit integer outputs.

In general, a pseudo-random sequence corresponding to a fixed key, k: $u_{k,0}, u_{k,1}, \ldots$, is obtained by starting with an initial state, s_0, in $\mathcal{S}$, and successive iteration of:

$$s_m = f(s_{m-1}) \qquad (4)$$
$$u_{k,n} = g_{k,n \bmod J}(s_{\lfloor n/J \rfloor}). \qquad (5)$$

The vast majority of PRNGs studied in the literature rely on careful design and analysis of the state transition function, f, and have output multiplicity $J = 1$. The output function, g, is usually trivial, consisting of nothing more than a linear mapping from an ordered finite set, $\mathcal{S}$, onto the set of real numbers, $[0, 1)$. We use the term *conventional* to refer to PRNGs that rely on a complicated f and a simple g. Of the 92 PRNGs surveyed by L'Ecuyer and Simard [24], 86 are conventional. All of the 45 PRNGs in version 1.14 of the GNU Scientific Library are conventional.

This paper revisits the underutilized design strategy [17, 37, 45, 48] of using a trivial f and building complexity into the output function through the composition of a simple selector, h_j, and a complex keyed bijection, b_k:

$$g_{k,j} = h_j \circ b_k.$$

In the rest of the paper, the state space, $\mathcal{S}$, will consist of p-bit integers, and since p is typically larger than the size of random numbers needed by applications, we have $J > 1$. The selector, h_j, simply chooses the jth r-bit block (with $r \leq \lfloor p/J \rfloor$) from its p-bit input. The state transition function, f, can be as simple as:

$$f(s) = (s + 1) \bmod 2^p.$$

Because f may be nothing more than a counter, we refer to such PRNGs as counter-based. The keyed bijection, b_k, is the essence of counter-mode PRNGs.

The two most important metrics for a PRNG are that its output be truly "random" and that it have a long period. Unfortunately, "randomness" is difficult to quantify and impossible to guarantee with absolute certainty [20, 22], hence the importance of extensive empirical testing. Once the statistical quality of a PRNG is established, secondary considerations such as speed, memory requirements, parallelizability, and ease of use determine whether a PRNG is used in practice.

2.1 Period of counter-based PRNGs

The period of any useful PRNG must be sufficiently long that the state space of the PRNG will not be exhausted by any application, even if run on large parallel machines for long periods of time. One million cores, generating 10 billion random numbers per second, will take about half an hour to generate 2^{64} random numbers, which raises doubts about the long-term viability of a single, unparararameterized PRNG with a periods of "only" 2^{64}. On the other hand, exhausting the state space of a multistreamable family of 2^{32} such generators, or a single generator with a period of 2^{128}, is far beyond the capability of any technology remotely like that in current computers. Thus, in Section 4, we focus on PRNGs with periods of 2^{128}, although the techniques described can easily support much longer periods.

More generally, because the bijection, f, clearly has period 2^p, and because each application of f gives rise to J output values, it follows that $J2^p$ is a period of the counter-based PRNGs in Section 4. For any two periods, P and Q, it is easy to show that $\gcd(P, Q)$ is also a period, and thus, if P is the smallest period, then any other period, Q (in particular, $J2^p$), must be a multiple of P. The number of factors of $J2^p$ is small and easily enumerated, and for each of the PRNGs in Section 4, we have empirically verified that none of the smaller factors of $J2^p$ is a period of the PRNG, and thus that $J2^p$ is, in fact, the smallest period.

Perhaps in response to a well-motivated concern over some short-period PRNGs in common use (e.g., the rand48() family of functions standardized by POSIX), the most popular PRNGs of recent years have had extraordinarily long periods. A Mersenne Twister [29], for instance, has a period of 2^{19937}, and the WELL family [35] has periods up to 2^{44497}. These periods completely eliminate concerns over exhausting the state space. On the other hand, these large state spaces require substantial amounts of memory and can be tricky to initialize properly [30]. If initialized carefully, the best super-long-period PRNGs work very well, but their period, in and of itself, does not make them superior in any practical way to PRNGs with merely long periods. A PRNG with a period of 2^{19937} is not substantially superior to a family of 2^{64} PRNGs, each with a period of 2^{130}.

2.2 Statistical tests and Crush-resistance

Long-standing practice is to subject PRNGs to an extensive battery of statistical tests, each of which is capable of a three-way distinction: *Failed*, *Suspicious*, and *Pass*. Each test relies on a statistic whose properties can be theoretically calculated under the assumption of uniform, independent, identically distributed (uiid) inputs. That statistic is measured on one or more sequences generated by the PRNG, and the distribution of the statistic is compared with the expected theoretical distribution. If the measurements would be highly unlikely under the null hypothesis of uiid inputs (i.e., the so-called p-value implies that the measurements would occur with probability less than, for instance, 10^{-10}), or if the result is "too good to be true" (i.e., a p-value between $1 - 10^{-10}$ and 1), the test is deemed to have *Failed*. On the other hand, if the measured statistic is unremarkable

(e.g., a p-value greater than 10^{-4} and less than $1 - 10^{-4}$), the test is deemed to have *Passed*. Intermediate p-values (between 10^{-10} and 10^{-4}) are considered *Suspicious*: they may simply be the result of bad luck, or they may be an indicator of a statistical failure right on the verge of detectability. Suspicious results are retried with more samples or a stronger (and perhaps costlier) statistic, until they either convincingly *Pass* or convincingly *Fail*.

A single *Failed* result is a clear sign that there is measurable structure in a PRNG's output. That structure *might* be reflected in the results of a simulation that uses the PRNG [12]. It is best to avoid PRNGs with such failures, especially since fast PRNGs exist that have no known failures [24], including those described in this work.

A single *Pass* result, on the other hand, tells us little—it says that a particular statistic is not sufficient to distinguish the output of the PRNG from bona fide samples of a random variable. Nevertheless, a large number of *Pass* results—and more importantly, the absence of any *Failed* results—on a wide variety of tests provides confidence (but not certainty) that the output of the PRNG is practically indistinguishable from a sequence of samples of an idealized random variable. Intuitively, if hundreds of very different statistics are all unable to detect any structure in the PRNG's output, then it is hoped that an application that uses the PRNG will be similarly oblivious to any structure.

Several statistical tests for PRNGs have been developed [4, 20, 26, 38], but the most comprehensive is TestU01 [24], which provides a consistent framework as well as batteries of tests that are supersets of the earlier suites, in addition to an assortment of tests from other sources that had not previously been incorporated into a battery. The pre-defined batteries in TestU01 include SmallCrush (10 tests, 16 p-values), Crush (96 tests, 187 p-values) and BigCrush (106 tests, 254 p-values). BigCrush takes a few hours to run on a modern CPU and in aggregate tests approximately 2^{38} random samples.[4]

2.2.1 Testing for inter-stream correlations

Statistical tests evaluate a single stream of random numbers, but additional care is needed when dividing the output of a PRNG into logical streams of random numbers. If the PRNG were perfectly random, any partitioning would be acceptable, but experience with conventional PRNGs suggests that fairly simple partitionings can expose statistical flaws in the underlying PRNG [16, 30]. It is thus important to verify that the distinct, non-overlapping streams created in this way are statistically independent.

An important advantage of counter-based PRNGs is the simplicity with which distinct parallel streams may be produced. We can test the independence of these streams by providing the TestU01 Crush batteries with an input sequence consisting of C values from the first logical stream, followed by C values from the next logical stream, up to the last, L^{th}, logical stream. The input then resumes for the next C values from the first logical stream, and so on. It is clearly impossible to run TestU01 with all possible logical sequences because of the enormous number of strategies by which an application can partition counters and keys into

logical streams. Nevertheless, it is imperative to check a representative sample of the kinds of logical streams that are expected to be encountered in practice: in particular, logical streams corresponding to different keys (the multistream approach), and corresponding to contiguous blocks of counters or strided and interleaved counters (the substream approach).

We chose representative samples of partitioning strategies as follows. First, we generated a sequence of triples, $(n_{key}, n_{blk}, n_{ctr})$, by traversing an $N_{key} \times N_{blk} \times N_{ctr}$ array in one of the six possible dimension orders.[5] We then generated the sequence of random values, u, for TestU01 from the sequence of triples with:

$$
\begin{aligned}
k &= k_0 + S_{key} * n_{key} \\
i &= i_0 + S_{blk} * n_{blk} + S_{ctr} * n_{ctr} \\
u &= b_k(i).
\end{aligned}
$$

The selection that takes three values at a time from each of 1000 key-based multistreams, for example, is captured by a dimension order from fastest to slowest of $(n_{ctr}, n_{key}, n_{blk})$ and $N_{key} = 1000$, $N_{blk} = \infty$, $N_{ctr} = 3$, arbitrary S_{key}, and $S_{blk} = 3, S_{ctr} = 1$. Taking seven values at a time from a large number of blocked substreams, with the same key and 2^{64} possible values in each block, is captured by a dimension order of $(n_{ctr}, n_{blk}, n_{key})$ and $N_{key} = 1$, $N_{blk} = \infty$, $N_{ctr} = 7$, arbitrary S_{key}, $S_{blk} = 2^{64}$, and $S_{ctr} = 1$.

Finally, we consider two *adversarial* counters that do not represent realistic use cases. Instead, these counters attempt to expose weaknesses in the underlying PRNG by sequencing through inputs with highly regular bit patterns. The *gray-coded counter* only changes one bit from one counter value to the next, while the *constant Hamming counter* keeps the same number of set and unset bits (e.g., 6 set and 122 unset) in each counter value. If, for example, a bijection tends to preserve the number of zero-bits from input to output, the constant Hamming counter will expose the flaw.

The input sequence to TestU01 is now characterized by dimension ordering, choice of adversarial counter, and a set of strides, offsets, and bounds. We have chosen a variety of such sequences that are representative of the kinds of blocking and striding that applications using counter-based PRNGs would actually employ. Depending on the size of the key and counter spaces, there are 89 to 139 such sequences. Among them are several that employ a constant key with strided counters, $S_{ctr} \in \{1, 4, 31415, 10000000, 2^{16}, 2^{32}, 2^{64}, 2^{96}\}$, as well as gray-coded and constant Hamming counters. Also included are similar strided and adversarial "key counter" sequences, in which the counter is fixed and only the key changes. Finally, possible correlations between keys and counters are explored by testing various combinations of N_{ctr}, N_{key}, N_{blk}, S_{ctr}, S_{key}, and S_{blk} from the set $\{1, 3, 4, 8, 16, 448, 4096, 31459, 2^{32}, 2^{64}\}$.

Each of the PRNGs proposed in Section 4 is tested with all three Crush batteries (SmallCrush, Crush and BigCrush) using all representative parallel sequences. All together, each PRNG is subject to at least $89 \times 212 = 18868$ tests with $89 \times 457 = 44786$ p-values. A PRNG that passes *all* these tests is called *Crush-resistant*. Except where noted, all of

---

[4]We modified the ClosePairs tests in Crush and BigCrush to generate each floating-point value from 64 bits of input. With 32-bit granularity, ClosePairs is susceptible to failure from detection of the discreteness of its inputs.

[5]The six dimension orders are the generalization of the two dimension orders in which a two-dimensional array can be traversed, commonly called *row major* and *column major*.

the PRNGs discussed in Section 4 are Crush-resistant.

As a practical matter, only a few conventional PRNGs pass even one complete battery of Crush tests. The multiple recursive generators, the multiplicative lagged Fibonacci generators, and some combination generators are reported to do so. On the other hand, many of the most widely used PRNGs fail quite dramatically, including all of the linear congruential generators, such as drand48() and the C-language rand(). The linear and general feedback shift register generators, including the Mersenne Twister, always fail the tests of linear dependence, and some fail many more. Similar statistical failures have been directly responsible for demonstrably incorrect simulation results in the past [12], suggesting that Crush-resistance is a valuable baseline for reducing the risk of such statistical failures.

3. CRYPTOGRAPHIC TECHNIQUES

The formulation of counter-based PRNGs (Eqn 3) is closely related to the operation of a block cipher in CTR mode [9]. Many of the most widely used and most thoroughly analyzed ciphers—including AES, DES [32], and a large number of less popular ciphers—are cryptographically strong bijections. The only difference between a counter-based PRNG and a CTR-mode block cipher is that in the PRNG, the output of the bijection is converted into a value in the output space rather than xor-ed with a block of plaintext. Furthermore, a necessary (but not sufficient) criterion for a cryptographically strong block cipher is that it satisfy the same statistical independence properties as a high-quality PRNG (see Section 2.2). Any block cipher that produced a sequence of values, $g_k(n)$, that was distinguishable from a stream of truly random, uiid samples of $\mathcal{S}$ would give an unacceptable "advantage" to an attacker and would be considered broken [1]. It is thus almost certain that any block cipher that withstands the extremely rigorous testing of the cryptographic community can be used as the bijection in Eqn 5 to produce a high-quality PRNG. In fact, the examples in the literature of counter-based PRNGs all use cryptographic block ciphers or cryptographic hashes [17, 24, 37, 45, 48]. Analyses of PRNGs based on AES, SHA-1, MD5 and ISAAC have shown that they all produce high-quality PRNGs in CTR mode [17, 24, 45]. PRNGs based on these block ciphers and hashes have not become popular in simulation largely because of performance—they are typically much slower than high-quality conventional PRNGs.

The property of producing highly random output from highly regular input is known as "diffusion" in the cryptographic literature, and is considered an essential property of any cryptographic cipher [39]. Diffusion can be quantified using the "avalanche criterion," which requires that any single-bit change in the input should result (on average) in a 50% probability change in each output bit.

In Section 4, we modify existing cryptographic block ciphers while still preserving their Crush-resistance, and we propose a new Crush-resistant bijection that is inspired by cryptographic techniques. To explore the trade-offs, it is helpful to be acquainted with some of the basic tools of modern cryptography.

Foremost among these is the idea of an *iterated cipher*. Most modern block ciphers consist of multiple iterations of simpler bijections, called "rounds." Each round introduces some diffusion, but by itself does not completely diffuse in-
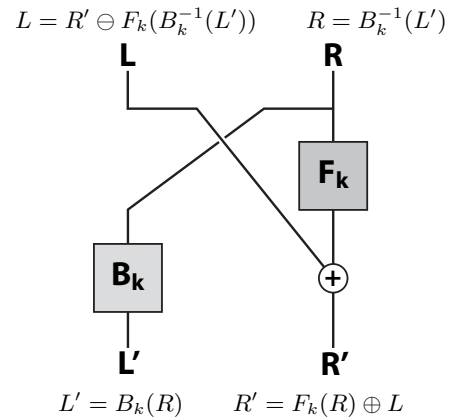

$$L = R' \ominus F_k(B_k^{-1}(L')) \qquad R = B_k^{-1}(L')$$

$$L' = B_k(R) \qquad R' = F_k(R) \oplus L$$

Figure 1: A generalized Feistel function maps $2p$-bit inputs (L,R) to $2p$-bit outputs, (L′,R′). Block F_k is an arbitrary key-dependent function mapping p-bit inputs to p-bit outputs, and block B_k is an arbitrary key-dependent bijection on the p-bit integers. (B_k is not normally present in the standard Feistel construction, that is, standard Feistel functions can be considered a special case in which B_k is the identity.) The operator $\oplus$ is a group operation on the p-bit integers (e.g., bitwise xor or addition modulo 2^p). A Feistel function is clearly a bijection because we can compute the unique inverse mapping using B_k^{-1} and $\ominus$, which are guaranteed to exist because B_k is a bijection and $\oplus$ is a group operator.

puts across outputs. Nevertheless, the composition of many simple rounds can result in a cryptographically strong bijection. For example, 128-bit AES has 10 rounds, DES has 16 rounds, and 256-bit Threefish has 72 rounds. As a "safety margin," the number of rounds employed for cryptographic use is significantly larger than the number of rounds that can be broken by the best known attack.

A second key insight from cryptography is the "Feistel function" (see Figure 1), first employed by the Lucifer system [43, 44], which provides a simple way to build a $2p$-bit bijection from an arbitrary (i.e., not bijective) p-bit, keyed function, F_k. Feistel functions are a common building block in modern ciphers.

Feistel functions allow us to work with 32-bit or 64-bit values (the word size of modern computer hardware) in fairly simple and CPU-friendly ways to produce bijections of 64-bit and 128-bit spaces.

Another important building block in modern cryptography is the "SP network" [10], illustrated in Figure 2. In an SP network, a highly diffusive bijection is obtained by iterating successive rounds of substitution, called "S-boxes" (narrow bijections, applied in parallel), and permutations, called "P-boxes" (wider, but highly structured mappings that simply permute the bits in a p-bit space). AES and Threefish, among many other block ciphers are structured as SP networks.

SP networks and Feistel functions are often combined in the same algorithm. For example, each round of DES is structured as a Feistel function, but each individual F_k is specified as an SP network.
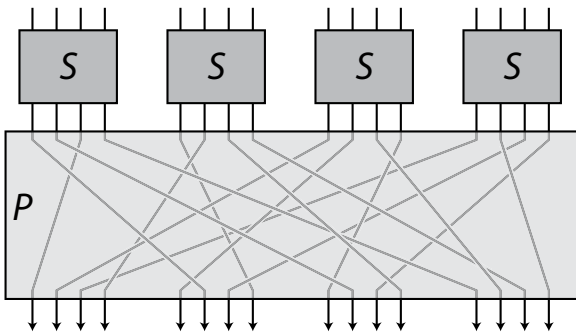
Figure 2: Each round of an SP network consists of a parallel series of narrow S-boxes, each of which is a bijection. The output of the S-boxes is then shuffled by a P-box. Some P-boxes (for instance, AES) are quite complex and troublesome for software implementation, but it is possible to design effective SP networks with P-boxes that work on word-wide chunks of bits and are very efficient in software (for instance, Threefish).

4. DESIGN OF EFFICIENT COUNTER-BASED PRNGS

Several approaches to the design of Crush-resistant bijections suggest themselves. First, an existing cryptographic block cipher could be used. It would be extremely surprising if any well-regarded cryptographic cipher failed any statistical tests. In general, cryptographic block ciphers are not used in PRNGs because they are slow without hardware support. A second approach is to trade strength for speed, which we discuss in Section 4.2. The resulting bijection will not be suitable for cryptography, but may be Crush-resistant and thus suitable for use in a PRNG. Finally, one can design an entirely new bijection using the principles of cryptography, but emphasizing statistical quality and speed.

4.1 Cryptographic PRNGs: AES, Threefish

The first family of counter-based PRNGs we consider is based on cryptographically strong block ciphers. These will produce random numbers whose quality should satisfy even the most cautious of users. We have confirmed the completely unsurprising result that these PRNGs are fully Crush-resistant.

The Advanced Encryption Standard, AES, has been considered for use in PRNGs before [17]. It is a 10-round iterated block cipher with a 128-bit counter-space and a 128-bit key-space.[6]

Beginning in 2010, high-performance CPUs from Intel (and in 2011, AMD) have supported the AES-NI instructions that dramatically improve the performance of an AES PRNG. Intel reports streaming performance of 1.3 cycles per byte (cpB) for cryptographic kernels using AES [14]. Using compiler intrinsics to invoke the AES instructions and gcc-4.5.2, we have measured an AES PRNG running at 1.7 cpB on a single core, faster than the Mersenne Twister in the C++0x standard library. This AES PRNG uses a pre-computed set of 11 "round keys," which require both time (approximately 100 cycles) to generate, and space (176 bytes) to store. We also have an implementation

that does on-the-fly generation of round keys, that requires no state space, and demands no setup time, but that has significantly lower throughput (7.3 cpB).

In contrast to AES, Threefish is a family of block ciphers designed with performance on modern 64-bit microprocessors in mind. Threefish is used in the Skein hash algorithm [11], one of six finalists in the SHA-3 [33] competition. The security properties of Skein are all proved in terms of claims about Threefish, so it seems reasonable to consider Threefish a cryptographically secure block cipher.

Threefish uses nothing more complicated than 64-bit addition, rotation, and xor, relying on a relatively large number of rounds (72 for Threefish-256) to achieve security. Although it uses an SP network, the permutation works on word-sized blocks and can be implemented in software with no data movement. Furthermore, modern processors can exploit instruction-level parallelism to compute several S-boxes simultaneously. Consequently, Threefish-256 is much faster than AES on platforms that lack the AES-NI instructions[7].

4.2 Fast PRNGs by reducing cryptographic strength: ARS, Threefry

Any modification of a cryptographic block cipher would make it unsuitable for its intended use in security, but it is reasonable for PRNG design to trade cryptographic strength for speed while still maintaining Crush-resistance. The most obvious simplifications include reducing the number of rounds, narrowing the data paths, and simplifying or eliminating the often complicated "key schedule," which is designed to keep the key hidden from a determined and resourceful attacker who is capable of choosing plaintext and observing the generated ciphertext.

We have investigated a few specific modifications of AES and Threefish, and call the resulting bijections ARS (Advanced Randomization System) and Threefry, respectively. ARS-N consists of N rounds of the AES round function, but uses a key schedule that follows a much simpler pattern inspired by the Weyl sequence.[8] Round keys are generated as follows, using conventional 64-bit addition on the separate high and low halves of the 128-bit ARS key:

$$\text{round\_key}_0 = \text{user\_key} \qquad (6)$$
$$\text{round\_key}_i = \text{round\_key}_{i-1} + \text{constant}.$$

The constants are fairly arbitrary. We have verified the Crush-resistance of ARS-5 (ARS-4 is not Crush-resistant) with the constants 0xBB67AE8584CAA73B (sqrt(3)-1) and 0x9E3779B97F4A7C15 (the golden ratio) for the upper and lower halves of the round key. ARS has superb performance as a PRNG on CPUs with AES-NI support; it is significantly faster than any other Crush-resistant PRNG (conventional or counter-based) on such CPUs.

Cryptographic strength can be traded for performance in Threefish in the same ways. Threefry-N×W-R follows the same pattern of S and P boxes as Threefish, but with R rounds, and N W-bit inputs and outputs. The key insertion

---

[6]Versions of AES also exist with key lengths of 192 and 256.

[7]The software implementation we used for AES is not the fastest known. AES has been reported to run at 10–15 cpB on a variety of platforms [2] (without AES-NI). It is thus likely that the software AES performance reported in Table 2 could be improved. Similarly, no attempt was made to vectorize our Threefish implementation.

[8]The Weyl sequence, $\{a \bmod 1, 2a \bmod 1, 3a \bmod 1, ...\}$, by itself, produces random numbers of poor quality, but it has been used successfully to overcome deficiencies in other generators [3].

	Round mod 8							
	0	1	2	3	4	5	6	7
Threefry-4×32	10	11	13	23	6	17	25	18
	26	21	27	5	20	11	10	20
Threefry-2×64	16	42	12	31	16	32	24	21

Table 1: Rotation constants used on each round of Threefry-N×W. The rotation constants are recycled after eight rounds.

schedule is the same as in Threefish, except that Threefry ignores the "tweak," an extra 128 bits of key-like input and associated arithmetic that seem superfluous for PRNG use.

For $W = 64$ and $N \geq 4$, Threefry's rotation constants are taken from Threefish. For other parameters, there are no published rotation constants for Threefish, so we generate Threefry's rotation constants (shown in Table 1) using the code provided with the Skein reference materials. Using these rotation constants, we have confirmed that Threefry-2×64-13, Threefry-4×32-12, and Threefry-4×64-12 are all Crush-resistant, and that Threefry-4×64-72 is identical to Threefish-256 with zero "tweak."

Threefry is the fastest Crush-resistant PRNG on CPUs without AES hardware support, and is among the fastest PRNGs on GPUs. Even Threefry-4×64-20, which has eight extra rounds of safety margin, is faster than any Crush-resistant conventional PRNG of which we are aware.

4.3 The Philox PRNG

Threefish, and by extension, Threefry, were designed to perform a relatively large number of very simple round functions. We now consider the alternative: an iterated bijection that achieves Crush-resistance with a smaller number of more complex round functions. The round function must not be too complex, because in order to be competitive with Threefry, it has to perform the full, multi-round bijection in 30–50 (preferably fewer) clock cycles. If we target 6–10 rounds (about half as many as Threefry), that gives about 5 clocks per round, which tightly bounds the design space.

If we restrict our attention to integer operations, the "obvious" way to quickly scramble the bits in a processor's registers is to use a multiply instruction.[9] The details of multiply instructions vary across architectures, but one way or another, every modern architecture offers single-instruction operations to do:

$$\text{mulhi}(a, b) \rightarrow \lfloor (a \times b)/2^W \rfloor$$
$$\text{mullo}(a, b) \rightarrow (a \times b) \bmod 2^W,$$

with W equal to the processor's word size. These multiply instructions have some interesting properties:

- For constant, odd M, $\text{mullo}(a, M)$ is a bijection.
- For constant M, a single bit flip in a causes an "avalanche" of changed bits in $\text{mullo}(a, M)$ and $\text{mulhi}(a, M)$. The avalanche occurs only in bits *more* significant than the flipped input bit in $\text{mullo}(a, M)$, and only in *less* significant bits in $\text{mulhi}(a, M)$.
- Integer multiplication takes between 2 and 10 cycles on common high-performance architectures, but can

generally be overlapped with other operations like addition, subtraction, and xor.
- On some architectures (notably x86 and x86-64) mulhi and mullo of the same arguments can be computed simultaneously by a single instruction.

Philox is an SP network with an overall structure similar to Threefry, but with an S-box that uses multiplication to provide diffusion. The Philox S-box is a generalized Feistel function (see Figure 1) using bitwise xor as the group operator, $\oplus$:

$$L' = B_k(R) = \text{mullo}(R, M)$$
$$R' = F_k(R) \oplus L = \text{mulhi}(R, M) \oplus k \oplus L.$$

For $N = 2$, the Philox-2×W-R bijection performs R rounds of the Philox S-box on a pair of W-bit inputs. For larger N, the inputs are permuted using the Threefish N-word P-box before being fed, two-at-a-time, into N/2 Philox S-boxes, each with its own multiplier, M, and key, k.[10] The $N/2$ multipliers are constant from round to round, while the $N/2$ keys are updated for each round according to the Weyl sequence defined in Eqn 6.

All that remains is to determine the multipliers. To find candidate multipliers, we first quantify the avalanche criterion (see Section 3) and then search for multipliers that maximize avalanche in a small number of rounds. Candidate multipliers found in this way are then subjected to the full suite of Crush tests to determine the minimum number of rounds that achieves Crush-resistance.

To quantify the avalanche of a function, f, define the differentials $\delta_{ij}(x)$ equal to one or zero according to whether the ith output bit of f changes when the jth input bit of x is changed. Then, for a large number D of random inputs, x_d, compute:

$$\Delta_{ij} = \sum_{d=1}^{D} \delta_{ij}(x_d).$$

If the avalanche criterion were satisfied, the values of Δ_{ij} would be independent samples of the binomial process with mean $D/2$ and variance $D/4$, a hypothesis that can be tested using a χ^2 statistic.

We search for multipliers that maximize the χ^2 statistic after four rounds[11] of Philox-N×W by starting with several thousand randomly chosen multipliers and refining them with an ad hoc strategy consisting of "point mutations" (one and two-bit modifications) genetic recombination (mixing fragments from good multipliers found so far) and steepest descent (keeping the values that maxmize the criterion). This procedure yielded the following multipliers: for Philox-4x32, 0xCD9E8D57 and 0xD2511F53; for Philox-2×64, 0xD2B74407B1CE6E93, and for Philox 4×64, 0xCA5A826395121157 and 0xD2E7470EE14C6C93. Full Crush-resistance was then verified using these multipliers with round counts of Philox-4×32-7, Philox-2×64-6, Philox-4×64-7.

Philox-4×32 relies on 32-bit multiplication, which is better-supported on current GPUs (and low-power, embedded CPUs). It is slightly slower than Philox-2×64 on

---

[9]This insight goes all the way back to the earliest random number generators, von Neuman's "middle square" [47] and Lehmer's linear congruential generators [25].

[10]For Philox-4xW, the permutation consists of swapping the R inputs between the two S-boxes.

[11]In all cases, the χ^2 values rule out the null hypothesis with high probability, indicating that four-round Philox fails the avalanche criterion and is inadequate as a PRNG.

Method	Max. input	Min. state	Output size	Intel CPU cpB	Intel CPU GB/s	Nvidia GPU cpB	Nvidia GPU GB/s	AMD GPU cpB	AMD GPU GB/s
Counter-based, Cryptographic									
AES(sw)	(1+0)×16	11×16	1×16	31.2	0.4	–	–	–	–
AES(hw)	(1+0)×16	11×16	1×16	**1.7**	**7.2**	–	–	–	–
Threefish (Threefry-4×64-72)	(4+4)×8	0	4×8	7.3	1.7	51.8	15.3	302.8	4.5
Counter-based, Crush-resistant									
ARS-5(hw)	(1+1)×16	0	1×16	0.7	17.8	–	–	–	–
ARS-7(hw)	(1+1)×16	0	1×16	**1.1**	**11.1**	–	–	–	–
Threefry-2×64-13	(2+2)×8	0	2×8	2.0	6.3	13.6	58.1	25.6	52.5
Threefry-2×64-20	(2+2)×8	0	2×8	2.4	5.1	15.3	51.7	30.4	44.5
Threefry-4×64-12	(4+4)×8	0	4×8	1.1	11.2	9.4	84.1	15.2	90.0
Threefry-4×64-20	(4+4)×8	0	4×8	**1.9**	**6.4**	15.0	52.8	29.2	46.4
Threefry-4×32-12	(4+4)×4	0	4×4	2.2	5.6	9.5	83.0	12.8	106.2
Threefry-4×32-20	(4+4)×4	0	4×4	3.9	3.1	15.7	50.4	25.2	53.8
Philox2×64-6	(2+1)×8	0	2×8	2.1	5.9	8.8	90.0	37.2	36.4
Philox2×64-10	(2+1)×8	0	2×8	4.3	2.8	14.7	53.7	62.8	21.6
Philox4×64-7	(4+2)×8	0	4×8	2.0	6.0	8.6	92.4	36.4	37.2
Philox4×64-10	(4+2)×8	0	4×8	3.2	3.9	12.9	61.5	54.0	25.1
Philox4×32-7	(4+2)×4	0	4×4	2.4	5.0	3.9	201.6	12.0	113.1
Philox4×32-10	(4+2)×4	0	4×4	3.6	3.4	**5.4**	**145.3**	**17.2**	**79.1**
Conventional, Crush-resistant									
MRG32k3a	0	6×4	1000×4	3.8	3.2	–	–	–	–
MRG32k3a	0	6×4	4×4	20.3	0.6	–	–	–	–
MRGk5-93	0	5×4	1×4	7.6	1.6	9.2	85.5	–	–
Conventional, Crushable									
Mersenne Twister	0	312×8	1×8	2.0	6.1	43.3	18.3	–	–
XORWOW	0	6×4	1×4	1.6	7.7	5.8	136.7	16.8	81.1

Table 2: **Memory and performance characteristics for a variety of counter-based and conventional PRNGs. Maximum input is written as (c+k)×w, indicating a counter type of width c×w bytes and a key type of width k×w bytes. Minimum state and output size are c×w bytes. Counter-based PRNG performance is reported with the minimal number of rounds for Crush-resistance, and also with extra rounds for "safety margin." Performance is shown in bold for recommended PRNGs that have the best platform-specific performance with significant safety margin. The multiple recursive generator MRG32k3a [21] is from the Intel Math Kernel Library and MRGk5-93 [23] is adapted from the GNU Scientific Library (GSL). The Mersenne Twister [29] on CPUs is std::mt19937_64 from the C++0x library. On NVIDIA GPUs, the Mersenne Twister is ported from the GSL (the 32-bit variant with an output size of 1×4 bytes). XORWOW is adapted from [27] and is the PRNG in the NVIDIA CURAND library.**

a single Xeon core, but it is significantly faster on a GPU. On an NVIDIA GTX580 GPU, Philox-4×32-7 produces random numbers at 202 GB per second per chip, the highest overall single-chip throughput that we are aware of for any PRNG (conventional or counter-based). On an AMD HD6970 GPU, it generates random numbers at 113 GB per second per chip, which is also an impressive single-chip rate. Even Philox-4×32-10, with three extra rounds of safety margin, is as fast as the non-Crush-resistant XORWOW[27] PRNG on GPUs.

We have only tested Philox with periods up to 2^{256}, but as with Threefish and Threefry, the Philox SP network is specified for widths up to 16×64 bits, and corresponding periods up to 2^{1024}. There is every reason to expect such wider forms also to be Crush-resistant.

5. COMPARISON OF PRNGS

When choosing a PRNG, there are several questions that users and application developers ask: Will the PRNG cause my simulation to produce incorrect results? Is the PRNG easy to use? Will it slow down my simulation? Will it spill

registers, dirty my cache, or consume precious memory? In Table 2, we try to provide quantitative answers to these questions for counter-based PRNGs, as well as for a few popular conventional PRNGs.[12]

All the counter-based PRNGs we consider are Crush-resistant; there is no evidence whatsoever that they produce statistically flawed output. All of them conform to an identical, naturally parallel API, so ease-of-use considerations are moot. All of them can be written and validated in portable C, but as with many algorithms, performance can be improved by relying on compiler-specific features or "intrinsics." All of them have periods in excess of 2^{128} and key spaces in excess of 2^{64}—well beyond the practical needs

---

[12]Tests were run on a 3.07 GHz quad-core Intel Xeon X5667 (Westmere) CPU, a 1.54 GHz NVIDIA GeForce GTX580 (512 "CUDA cores") and an 880 Mhz AMD Radeon HD6970 (1536 "stream processors"). The same source code was used for counter-based PRNGs on all platforms and is available for download. It was compiled with gcc 4.5.3 for the Intel CPU, the CUDA 4.0.17 toolkit for the NVIDIA GPU and OpenCL 1.1 (AMD-APP-SDK-v2.4) for the AMD GPU. Software AES used OpenSSL-0.9.8e. The MRG generators were from version 10.3.2 of the Intel Math Kernel library and version 1.12 of the GNU Scientific Library.

of computers in the forseeable future.

On-chip memory is a crucial resource on modern hardware, and ignoring it can lead to sub-optimal design choices. The highest-performing conventional PRNGs may only get "up to speed" when producing thousands of random values in a batch, but applications may only consume a few random values at a time in each thread. The output of such PRNGs can, of course, be buffered into large chunks, but this entails a substantial memory cost that may limit parallelism or performance.

To quantify the memory requirements, we distinguish between the input size, the state size and the output granularity of each PRNG. To some extent, these sizes are governed by trade-offs in specific implementations rather than the fundamental algorithm. For example, the AES bijection can be implemented either with a pre-computed set of round keys (which requires an 11×128-bit state) or with on-the-fly round keys (which is slower, but requires no state). In Table 2, we report the minimum state size and maximum allowable input size for each invocation of the measured implementations of some selected PRNGs. Any user of these PRNGs will incur the memory costs of the reported amount of state, and must provide counter and key values of the specified input size on every invocation. Some users may find it convenient, however, to zero-pad a smaller input if the full period length or key space is not needed. One can also use global memory or even compile-time constants for some portion of the input.

The output granularity of a counter-based PRNG is typically the size of the counter. That is, each invocation of the underlying API generates random data of the same size as the counter, so an application that needs a smaller amount of random data will be required to buffer the excess output somewhere. Conventional PRNGs have very different granularity characteristics. In some cases (e.g., GSL, C++0x), the granularity is hidden by the library—single values are returned, regardless of the internal workings of the PRNG. In other cases (e.g., the Intel Math Kernel Library), the output granularity is chosen by the user, but the PRNG's performance depends very strongly on the choice [18].

We report single-chip PRNG performance in gigabytes per second.[13] This metric is chip-dependent, but provides data on the relative computational throughput of some current high-performance processor chips. We also report PRNG performance in single-core cpB—that is, the amortized number of clock cycles required for a single core to produce a random byte. Single-core cpB allows one to assess the relative cost of random number generation compared to the code that uses the random numbers without the distraction of clock-speed or number of cores.

At 1 to 4 cpB, the raw speed of Crush-resistant PRNGs is past the point of diminishing returns because very few applications are demanding enough to notice one or two extra clock cycles per byte of random data. It is thus advisable to trade a few cycles of performance for an additional, but unquantifiable, margin of safety conferred by adding rounds. (The additional safety is unquantifiable because we have no evidence of any statistical flaws that could be improved.)

In Table 2, we report performance for the variants of ARS,

---

[13]When measuring performance, we accumulate a running sum of the random numbers generated, but do not write them to memory on either CPUs or GPUs, to avoid memory or bus bandwidth effects.

Philox, and Threefry with the minimum number of rounds required for Crush-resistance, as well as variants with additional rounds as a safety margin (ARS-7, Philox with 10 rounds and Threefry with 20 rounds). We favor use of the latter variants, with the extra safety margin, since the performance penalty is quite small.

No clear winner emerges from Table 2. One method for approaching the choice is as follows. If an application can rely on the presence of the AES-NI instructions and can tolerate 176 bytes of key state per multi-stream, then AES is highly recommended: it is fast, thoroughly scrutinized and cryptographically strong. If storage of pre-computed keys is too costly, or if the last iota of performance is critical, ARS-7 is advised. Without AES-NI, we recommend Threefry-4×64-20 on CPUs and Philox-4×32-10 on GPUs. They are portable, indistinguishable in terms of statistical quality (both are Crush-resistant with a substantial safety margin), and faster than any Crush-resistant conventional PRNG.

5.1 An API for parallel random number generation

It is easy to parallelize a counter-based PRNG using either or both the multistream and substream approaches. In the multistream approach, the application assigns a unique key, k, to each computational unit and then the units generate their own sets of random numbers in parallel by incrementing their own counters n, and computing $b_k(n)$. In the substream approach, the application partitions the counter space among the computational units, and each unit generates the random numbers, $b_k(n)$, where n is in its set of assigned counters and k is the same across all units. Partitioning the integers is easy. Strategies like blocking (task t gets the N integers from $t \times N$ to $[t+1] \times N$) and striding (task t gets the integers, n, for which $n \bmod T = t$) are simple to code and require almost no overhead.

In contrast to conventional PRNGs, where advancing the state using a carefully crafted transition algorithm is the library's essential role, an API that advances the state of counter-based PRNGs would obscure more than it illuminated. Instead, a natural API for counter-based PRNGs provides the keyed bijection, b_k, but leaves the application in complete control of the key, k, and counter, n. Such control allows the application to manage parallelism in any way that is most convenient or effective, and makes available a wide range of strategies for producing essentially unlimited independent streams of numbers with little or no memory overhead. If, for example, an application with a large number of objects already has stable integer identifiers, i, for those objects, as well as a monotonically advancing notion of "time" (T), and needs a random number per object for each value of T, it could concatenate the object identifier, i, and time, T, into a p-bit counter, n, from which it can obtain object-specific, time-dependent random numbers by computing $b_k(n)$; this method amounts to a substream approach. Alternatively, an application could select the object identifier, i, as the key and the time, T, as the counter, and use $b_i(T)$ to get streams of random numbers for each object (i.e., a multistream approach). In either of these cases, no extra memory is needed to store state for these streams of random numbers.

Most applications require the capability for users to specify "seed" values for the PRNG. The purpose of the seed is

to allow a user to run a variety of different simulations with otherwise identical parameters—for example, to sample different regions of a search space. Counter-based PRNGs easily accomodate seeds. One can simply reserve 32 bits in the key or the counter for a 32-bit seed common to all threads in a parallel application. With 64 bits (or more) of key space and 128 bits of counter space, there remains plenty of space for parallel multistreaming or substreaming.

An even more important aspect of counter-based PRNGs is that no synchronization or communication is needed across different parallel streams of random numbers, which allows great flexibility in application parallelization strategy. A particle simulation, for example, may be parallelized so that forces on particles that happen to be near the boundary between some processors' domains will be computed redundantly on multiple processors to avoid costly inter-processor communication. If the simulation also requires random forces, counter-based PRNGs can easily provide the same per-atom random forces on all the processors with copies of that atom by using the atom identifier and timestep value as described above. Such redundant computation is difficult with a conventional PRNG unless a unique per-particle PRNG state is maintained, which can be very costly, or in some cases impossible. Avoiding the need for communication and synchronization removes a significant bottleneck to scaling such particle simulations to very large numbers of cores.

The one important precondition that the application must maintain when managing keys and counters is that (key, counter) tuples must never be *improperly* re-used, since an inadvertent re-use of the same tuple will result in exactly the same random number, with potentially dire consequences for simulation accuracy [5, 42]. This precondition is not particularly onerous, since many naturally unique sources for keys and counters exist in applications, and even if there are no appropriate simulation state variables, a simple global counter is trivial to maintain. In fact, the error identified by [5] arose because of a failure to properly checkpoint and restore the PRNG's state. If the PRNG were driven by the logical simulation state, such errors would be far less common.

Our experiments with counter-based PRNGs are all written using a common API with C and C++ bindings. All functions are "pure," with no internal state, so bindings for other languages, including functional languages, would be easy to devise. The C bindings provide three typedefs and two functions for each implemented PRNG. The typedefs specify the counter type and two key types. A `keyinit` function transforms a *user key* into the key that that is used as an argument, along with a counter, to the `bijection` function. The counter and userkey types are structs containing a single array member—simple enough for compilers to aggressively optimize, and easy for application developers to work with. In most cases `keyinit` simply copies its input to its output, but in some cases (e.g., AES) a non-trivial calculation is required to expand the user key into an opaque internal key.

The C++ API enhances the userkey and counter types with member functions, friends, and typedefs so that they closely resemble the std::array objects standardized by C++0x. It also provides bijections as functor classes with member typedefs and an overloaded `operator()` for the bijection itself.

For the benefit of applications that expect a conventional API, the C++ API also provides a template wrapper that maps Bijection classes into fully standardized, conventional C++0x "Random Number Engines." These engines can then be used as standalone sources of random numbers, or they can be used in conjunction with other library utilities such as random distributions. Wrapping the bijections with an adapter that conforms to another API (e.g., GSL) would also be straightforward.

The C and C++ APIs are implemented entirely in header files, which facilitates porting and allows compilers to perform significantly better optimization. In particular, constant folding has a significantly beneficial effect if the bijections are called with constants for the number of rounds. The same headers can be compiled into binaries targeted at CPUs (using a standard compiler) or GPUs (using either CUDA or OpenCL). The timings for counter-based PRNGs in Table 2 were obtained using the same underlying source code and machine-specific compilation systems.

6. CONCLUSIONS

The keyed counter-based PRNGs presented in this paper can be naturally and easily parallelized over large numbers of cores, nodes, or logical software abstractions (particles, voxels, etc.) within large-scale simulations. These PRNGs are fast, require little or no state, and are easily initialized. They have long periods and have passed the most comprehensive set of statistical tests we have found (including tests that many popular, conventional PRNGs fail).

Implementations of several of the PRNGs described in this paper, including test vectors, API bindings, and example programs, are available in source form for download from: `http://deshawresearch.com/resources_random123.html` A single set of header files implements the API in C, C++98, C++0x, CUDA and OpenCL environments. Although a conventional C++0x "random engine" is provided, we find the use of the native API much better suited to highly parallel applications because it permits on-the-fly random number generation from unique application-level logical state variables.

Among our counter-based PRNGs, the Philox and Threefry families include the fastest PRNGs of which we are aware for GPUs and CPUs without AES-NI support. Our ARS family contains the fastest PRNG of which we are aware for modern CPUs with AES-NI support. Counter-based PRNGs using the full AES and Threefish encryption algorithms are slower, but are built on ciphers that have withstood the scrutiny of the cryptographic community. The best news about counter-based PRNGs may well be that the ones presented here are just the tip of the iceberg. Many more statistically strong, high-performance, low-memory, long-period, parallelizable, counter-based PRNGs remain undiscovered or unrecognized.

Because all the counter-based PRNGs described above require fewer clock ticks than a single cache miss or memory access, considerations other than raw speed seem more relevant—particularly, statistical quality, ease of parallelization, safe seeding, convenience of key or counter management, and number of bytes of random data returned on each invocation. Counter-based PRNGs excel on all these fronts.

7. REFERENCES

[1] M. Bellare and P. Rogaway. Pseudorandom functions. In *Introduction to Modern Cryptography*. UCSD CSE 207 Online Course Notes, 2011. Chap 3. `http://cseweb.ucsd.edu/~mihir/cse207/w-prf.pdf`.

[2] D. J. Bernstein and P. Schwabe. New AES software speed records. In D. R. Chowdhury and V. Rijmen, editors, *Progress in Cryptology - INDOCRYPT 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 322–336, Berlin, 2008. Springer-Verlag.

[3] R. P. Brent. Some long-period random number generators using shifts and xors. *ANZIAM Journal*, 48:C188–C202, 2007.

[4] R. G. Brown. Dieharder: A random number test suite. `http://phy.duke.edu/~rgb/General/dieharder.php`.

[5] D. S. Cerutti, R. Duke, P. L. Freddolino, H. Fan, and T. P. Lybrand. Vulnerability in popular molecular dynamics packages concerning Langevin and Andersen dynamcs. *J. Chem. Theory*, 4:1669–1680, 2008.

[6] P. Coddington. Random number generators for parallel computers. Technical Report 13, Northeast Parallel Architecture Center, 1997.

[7] A. De Matteis and S. Pagnutti. Parallelization of random number generators and long-range correlations. *Numer. Math.*, 53:595–608, August 1988.

[8] A. De Matteis and S. Pagnutti. Long-range correlations in linear and non-linear random number generators. *Parallel Computing*, 14:207–210, 1990.

[9] M. Dworkin. Recommendation for block cipher modes of operation, methods and techniques. NIST Special Publication 800-38A. National Institute of Standards and Technology (NIST), 2001.

[10] H. Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, 1973.

[11] N. Ferguson, S. Lucks, B. Schneier, B. Whiting, M. Bellare, T. Kohno, J. Callas, and J. Walker. The Skein hash function family. `http://www.schneier.com/skein.pdf`, 2010.

[12] A. M. Ferrenberg, D. P. Landau, and Y. J. Wong. Monte Carlo simulations: Hidden errors from "good" random number generators. *Phy. Rev. Lett.*, 69:3382–3384, 1992.

[13] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors; Volume 1: General Techniques and Regular Problems*. Prentice-Hall, 1988.

[14] S. Gueron. Intel Advanced Encryption Standard (AES) instructions set. Technical report, Intel, 2010.

[15] P. Hellekalek. Don't trust parallel Monte Carlo! In *Proc. 12th Workshop on Parallel and Distributed Simulation*, PADS '98, pages 82–89, Washington, D.C., 1998. IEEE Computer Society.

[16] P. Hellekalek. Good random number generators are (not so) easy to find. *Math. Comput. Simul.*, 46:485–505, June 1998.

[17] P. Hellekalek and S. Wegenkittl. Empirical evidence concerning AES. *ACM Trans. Model. Comput. Simul.*, 13:322–333, October 2003.

[18] Intel. Vector Statistical Library (VSL) performance data. `http://software.intel.com/sites/products/documentation/hpc/mkl/vsl/vsl_performance_data.htm`.

[19] M. H. Kalos and P. A. Whitlock. *Monte Carlo Methods*. Wiley-VCH, 2nd edition, 2008.

[20] D. E. Knuth. *The Art of Computer Programming, Volume 2 (3rd ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1997.

[21] P. L'Ecuyer. Combined multiple recursive random number generators. *Operations Research*, 44(5):816–822, 1996.

[22] P. L'Ecuyer. Random number generation. In J. E. Gentle, W. Haerdle, and Y. Mori, editors, *Handbook of Computational Statistics*, pages 35–70. Springer-Verlag, Berlin, 2004. Chapter II.2.

[23] P. L'Ecuyer, F. Blouin, and R. Couture. A search for good multiple recursive random number generators. *ACM Trans. Model. Comput. Simul.*, 3(2):87–98, 1993.

[24] P. L'Ecuyer and R. Simard. TestU01: A C library for empirical testing of random number generators. *ACM Trans. Math. Softw.*, 33, August 2007.

[25] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proc. 2nd Symp. on Large-Scale Digital Calculating Machinery*, pages 141–146. Harvard University Press, 1949.

[26] G. Marsaglia. DIEHARD: A battery of tests of randomness. `http://stat.fsu.edu/~geo/diehard.html`.

[27] G. Marsaglia. Xorshift RNGs. *J. Stat. Soft.*, 8:1–6, 2003.

[28] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: A scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software*, 26:436–461, 2000.

[29] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8:3–30, January 1998.

[30] M. Matsumoto, I. Wada, A. Kuramoto, and H. Ashihara. Common defects in initialization of pseudorandom number generators. *ACM Trans. Model. Comput. Simul.*, 17, September 2007.

[31] E. H. Mckinney. Generalized birthday problem. *The American Mathematical Monthly*, 73(4):385–387, April 1966.

[32] National Bureau of Standards. Data Encryption Standard. FIPS PUB 46-3, 1977.

[33] National Institute of Standards and Technology. Cryptographic hash algorithm competition website. `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.

[34] National Institute of Standards and Technology. Advanced Encryption Standard (AES). FIPS PUB 197, 2001.

[35] F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Trans. Math. Softw.*, 32:1–16, March 2006.

[36] S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Commun. ACM*, 31:1192–1201, October 1988.

[37] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and

B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, Cambridge, 2nd edition, 1992.

[38] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, M. Levinson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Special Publication 800-22 Revision 1a, NIST, April 2010.

[39] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28:656–715, 1949.

[40] D. E. Shaw, R. O. Dror, J. K. Salmon, J. P. Grossman, K. M. Mackenzie, J. A. Bank, C. Young, M. M. Deneroff, B. Batson, K. J. Bowers, E. Chow, M. P. Eastwood, D. J. Ierardi, J. L. Klepeis, J. S. Kuskin, R. H. Larson, K. Lindorff-Larsen, P. Maragakis, M. A. Moraes, S. Piana, Y. Shan, and B. Towles. Millisecond-scale molecular dynamics simulations on Anton. In *Proc. Conf. on High Performance Computing, Networking, Storage and Analysis*, SC09, pages 39:1–39:11, New York, NY, 2009. ACM.

[41] D. E. Shaw, P. Maragakis, K. Lindorff-Larsen, S. Piana, R. O. Dror, M. P. Eastwood, J. A. Bank, J. M. Jumper, J. K. Salmon, Y. Shan, and W. Wriggers. Atomic-level characterization of the structural dynamics of proteins. *Science*, 330:341–346, 2010.

[42] D. J. Sindhikara, S. Kim, A. F. Voter, and A. E. Roitberg. Bad seeds sprout perilous dynamics: Stochastic thermostat induced trajectory synchronization in biomolecules. *J. Chem. Theory and Comp.*, 5(6):1624–1631, 2009.

[43] J. L. Smith. The design of Lucifer, a cryptographic device for data communications. IBM Research Report RC3326, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA, Apr. 1971.

[44] A. Sorkin. Lucifer, a crytographic algorithm. *Cryptologia*, 8:22–35, 1984.

[45] S. Tzeng and L.-Y. Wei. Parallel white noise generation on a GPU via cryptographic hash. In *Proc. 2008 Symp. on Interactive 3D graphics and games*, I3D '08, pages 79–87, New York, NY, 2008. ACM.

[46] S. Ulam, R. Richtmeyer, and J. von Neumann. Statistical methods in neutron diffusion. Technical Report LAMS-551, Los Alamos Scientific Laboratory, April 1947.

[47] J. von Neuman. Various techniques used in connection with random digits. In A. Householder, G. Forsythe, and H. Germond, editors, *Monte Carlo Method, Applied Math Series, Volume 11*, pages 36–38. National Bureau of Standards, 1951.

[48] F. Zafar, M. Olano, and A. Curtis. GPU random numbers via the Tiny Encryption Algorithm. In *Proc. Conf. High Performance Graphics*, HPG '10, pages 133–141, Aire-la-Ville, Switzerland, 2010. Eurographics Association.