Batched Ranged Random Integer Generation

Nevin Brackett-Rozinsky[1] and Daniel Lemire[1]

[1]Universite TELUQ

August 27, 2024

Abstract

Pseudorandom values are often generated as 64-bit binary words. These random words need to be converted into ranged values without statistical bias. We present an efficient algorithm to generate multiple independent uniformly-random bounded integers from a single uniformly-random binary word, without any bias. In the common case, our method uses one multiplication and no division operations per value produced. In practice, our algorithm can triple the speed of unbiased random shuffling for small to moderately large arrays.

RESEARCH ARTICLE

Batched Ranged Random Integer Generation

Nevin Brackett-Rozinsky[1] | **Daniel Lemire**[2]

[1]Limerick, Maine, USA

[2]Data Science Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada

Correspondence
Daniel Lemire, Data Science Research Center, Université du Québec (TELUQ), Montreal, Quebec, H2S 3L5, Canada
Email: daniel.lemire@teluq.ca

Funding information
Natural Sciences and Engineering Research Council of Canada, Grant Number: RGPIN-2024-03787

Pseudorandom values are often generated as 64-bit binary words. These random words need to be converted into ranged values without statistical bias. We present an efficient algorithm to generate multiple independent uniformly-random bounded integers from a single uniformly-random binary word, without any bias. In the common case, our method uses one multiplication and no division operations per value produced. In practice, our algorithm can triple the speed of unbiased random shuffling for small to moderately large arrays.

KEYWORDS
Random number generation, Rejection method, Randomized algorithms

1 | INTRODUCTION

Random numbers are useful in many computer programs. Most programming languages provide a method to generate uniformly random or pseudorandom integers in the range $[0, 2^L)$ for some L, commonly 64. We refer to such numbers as L-bit random words, and the functions which produce them as *random number generators*. Pseudorandom generators execute algorithms to produce a sequence of numbers that approximate the properties of random numbers, starting from a given *seed* (typically a few bytes). There are a variety of efficient techniques to generate high-quality pseudorandom binary words, such as the Mersenne Twister [1], linear congruential generators [2, 3, 4, 5, 6], PCG [7], and so forth [8, 9]. Although they do not produce truly random outputs, many pseudorandom generators can pass rigorous statistical tests, and are considered to be random in practice [10].

Random binary words are readily available, yet applications often require uniformly random integers from other bounded ranges, such as $[0, b)$ for some b. We refer to these numbers as bounded random integers, or *dice rolls*, and we focus on the case where $0 < b \leq 2^L$. We are particularly interested in applications which require multiple dice rolls, and where b may be chosen dynamically rather than known at compile time, such as when shuffling arrays or selecting random samples [11]. For example, we consider the Fisher-Yates random shuffle described by Knuth [12, 13] and by

Durstenfeld [14], which we restate here in Algorithm 1.

Algorithm 1 — Fisher-Yates random shuffle

Require: Source of uniformly random integers in bounded ranges
Require: Array A made of n elements indexed from 0 to $n-1$
Ensure: All $n!$ permutations of A are equiprobable

1: **for** $i = n-1, \ldots, 1$ **do**
2: $\quad j \leftarrow$ random integer in $[0, i]$
3: $\quad$ exchange $A[i]$ and $A[j]$

Algorithms such as random shuffling are commonplace in simulations and other important applications [15, 16, 17, 18, 19, 20, 21]. Accordingly, there has been much work done on parallelizing random permutations [22, 23, 24, 25, 26]. In the present work, we demonstrate that batching dice rolls can improve the performance of shuffles and similar algorithms, by reducing the number of calls to a random number generator.

Perhaps surprisingly, the computational cost of converting binary words into ranged integers can be critical to good performance. For example, Lemire [27] showed that by simplifying the conversion to avoid most division operations, the practical performance of a random shuffle could be made up to eight times faster. We aim to show that we can further multiply the performance in some cases, through the use of batched dice rolls.

Our main theoretical result is in § 4, where we present and prove the correctness of an algorithm to generate multiple independent bounded random integers from a single random binary word, using in the common case only one multiplication and zero division operations per die roll. The method is based on an existing algorithm due to Lemire that generates one such number at a time, which we summarize in § 3. Our proof uses mixed-radix notation as described in § 2.

In § 5 we demonstrate how our theoretical result can be implemented as a practical algorithm, in § 6 we apply it to the task of shuffling an array, and in § 7 we show the results of our experiments, with timing measurements illustrating the speed of array shuffling with and without batched dice rolls. Finally in § 8 we summarize our results.

1.1 | Mathematical notation

We only consider non-negative integers. We use "$\otimes$" to denote full-width multiplication: $a \otimes b = (x, y)$ means x and y are integers such that $2^L x + y = ab$, and $0 \le y < 2^L$. On x64 systems, a full-width multiplication requires only a single instruction, while ARM systems provide full-width multiplication with two instructions.

We use "$\div$" to denote integer division: $a \div b = \lfloor a/b \rfloor$ is the greatest integer less than or equal to a/b. We use "mod" to denote the Euclidean remainder: $(a \bmod b) = a - b \cdot (a \div b)$. Because we use only non-negative integers, we have $0 \le (a \bmod b) < b$. Note that division and remainder instructions are often slow in practice, and may have a latency of 18 cycles compared to merely 3 cycles for a multiplication on a recent Intel processor (e.g. Ice Lake) [28].

We use pi notation "$\prod$" to denote products, and we omit the bounds when they can be inferred from context. Thus if b_i is defined for each i from 1 through k, then $\prod b_i = b_1 b_2 \cdots b_k$. We also use sigma notation "$\sum$" to denote sums, and an underlined superscript to denote the falling factorial: $n^{\underline{k}} = n!/(n-k)! = n(n-1)\cdots(n-(k-1))$.

2 | MIXED-RADIX NUMBERS

We use mixed-radix notation in the proof of our main result. Mixed-radix notation is a positional number system in which each digit position has its own base. This contrasts with decimal notation where every digit is in base 10, or binary where every digit is in base 2. Mixed-radix numbers are allowed, but not required, to have a different base for each digit. Each base is a positive integer, and each digit is a non-negative integer smaller than its base.

We denote both the bases and digits of a mixed-radix number with ordered tuples beginning with the most-significant digit, and use context to distinguish them. For example, a 2-digit mixed-radix number in base (b_1, b_2) whose digits are (a_1, a_2) represents the value $a = a_1 b_2 + a_2$. A k-digit mixed-radix number in base $(b_1, b_2, \ldots, b_k)$ can represent all the integers from 0 through $b-1$, where $b = b_1 b_2 \cdots b_k$, and this representation is unique. If its digits are $(a_1, a_2, \ldots, a_k)$ then it represents the value:

$$a = \sum_{i=1}^{k} \left(a_i \prod_{j>i}^{k} b_j \right) = a_1(b_2 b_3 \cdots b_k) + a_2(b_3 b_4 \cdots b_k) + \cdots + a_{k-1} b_k + a_k$$

Given a non-negative integer $a < \prod b_i$, its mixed-radix representation can be obtained by taking the remainders of successive quotients when dividing by $b_k, b_{k-1}, \ldots, b_1$, meaning all the b_i in reverse order.

Lemma 1 *If a uniformly random integer a in $[0, b)$ is written as a mixed-radix number in base $(b_1, b_2, \ldots, b_k)$, where $b = \prod b_i$, then its digits $(a_1, a_2, \ldots, a_k)$ are independent and uniformly random integers in the ranges $0 \le a_i < b_i$.*

Proof Every possible sequence of digits is equally likely, so each a_i takes all integer values in $[0, b_i)$ with uniform probability regardless of the values of the other digits.

3 | EXISTING ALGORITHMS

There are many ways to generate bounded random integers from random bits, and it is nontrivial to do so efficiently. Several widely-used algorithms are described by Lemire [27], including a then-novel strategy to avoid expensive division operations. This method has now been adopted by several major systems: GNU libstdc++, Microsoft standard C++ library, the Linux kernel, and the standard libraries of the Go, Swift, Julia, C# and Zig languages.

It works as follows. Let r be a uniformly random integer in $[0, 2^L)$, and $[0, b)$ the desired target range with $0 < b \le 2^L$. Perform the full-width multiplication $b \otimes r = (x, y)$. If r is picked uniformly at random and we have $y \ge (2^L \bmod b)$—which we call *Lemire's criterion*—then x is uniformly random in $[0, b)$. Otherwise, try again with a new r. That is, apply the rejection method [29].

In the common case, Lemire's method uses one L-bit random word and one multiplication per die roll. Sometimes we must compute the remainder $(2^L \bmod b) = ((2^L - b) \bmod b)$ which may require a division instruction. Thankfully, we can often avoid the division. Because $(2^L \bmod b) < b$, we only need to compute $(2^L \bmod b)$ when $y < b$. When b is much smaller than 2^L the division is often avoided, and when required it is computed at most once per die roll.

3.1 | Batched dice rolls

One way to generate bounded random numbers in batches is that, to simulate rolling dice with b_1 and b_2 sides, a single die with $b_1 b_2$ sides is rolled instead. The resulting number is uniformly random in $[0, b_1 b_2)$, and the desired

dice rolls can be obtained by taking its remainder and quotient upon dividing by b_1. These values are independent and uniformly random integers in $[0, b_1)$ and $[0, b_2)$ respectively, and the method generalizes to more dice by taking successive remainders of quotients upon dividing by each b_i. This strategy uses fewer random bits than rolling each die separately, however it is only of theoretical interest because division operations are slow in practice.

Our approach works differently. Rather than extract the values via division, we instead build them up through multiplication. Specifically, we extend Lemire's method in order to generate multiple bounded random integers from a single random word, with zero division operations in the common case.

4 | MAIN RESULT

Theorem 1 *Let r_0 be an L-bit random word, meaning r_0 is a uniformly random integer in $[0, 2^L)$. Let $(b_1, b_2, \dots, b_k)$ be positive integers with $b = \prod b_i \leq 2^L$. Starting with r_0 and b_1, for each i from 1 to k, perform the full-width multiplication $b_i \otimes r_{i-1}$ and set a_i to the most significant L bits and r_i to the least significant L bits of the $2L$-bit result:*

- $(a_1, r_1) \leftarrow b_1 \otimes r_0$
- $(a_2, r_2) \leftarrow b_2 \otimes r_1$
- $\vdots$
- $(a_k, r_k) \leftarrow b_k \otimes r_{k-1}$

If r_0 is picked uniformly at random and $r_k \geq (2^L \bmod b)$ is satisfied, then the a_i are independent and uniformly random integers in the ranges $0 \leq a_i < b_i$.

Proof We will first show that when the conditions of the theorem are met, if the resulting values $(a_1, a_2, \dots, a_k)$ are interpreted as the digits of a mixed-radix number a in base $(b_1, b_2, \dots, b_k)$, then a is a uniformly random integer in $[0, b)$. Once this is established, we will invoke Lemma 1.

For each i from 1 to k, the full-width product $b_i \otimes r_{i-1} = (a_i, r_i)$ means $b_i r_{i-1} = 2^L a_i + r_i$. Since both r_{i-1} and r_i are in $[0, 2^L)$, this implies $0 \leq a_i < b_i$. So each a_i is a valid mixed-radix digit for base b_i, and a is well-defined. Let c_i be the value obtained by truncating a to its first i digits. In other words, c_i is the value represented by the mixed-radix number $(a_1, a_2, \dots, a_i)$ in base $(b_1, b_2, \dots, b_i)$. Thus $c_1 = a_1$, and $c_i = b_i c_{i-1} + a_i$ for $1 < i \leq k$.

Let $b_1 b_2 \cdots b_i$ be the product of $b_1, b_2, \dots, b_i$. We claim that $(b_1 b_2 \cdots b_i) \otimes r_0 = (c_i, r_i)$, and we prove it by finite induction on i. The claim is equivalent to $r_0(b_1 b_2 \cdots b_i) = 2^L c_i + r_i$, which is true for $i = 1$. If $1 < i \leq k$ we use the inductive hypothesis that the claim is true for $i-1$, in order to prove it for i. Thus, we have:

$$\begin{aligned}
r_0(b_1 b_2 \cdots b_i) &= r_0(b_1 b_2 \cdots b_{i-1}) b_i \\
&= (2^L c_{i-1} + r_{i-1}) b_i && \text{(inductive hypothesis)} \\
&= 2^L b_i c_{i-1} + b_i r_{i-1} && \text{(distributivity)} \\
&= 2^L b_i c_{i-1} + 2^L a_i + r_i && \text{(definition of } a_i \text{ and } r_i\text{)} \\
&= 2^L (b_i c_{i-1} + a_i) + r_i && \text{(distributivity)} \\
&= 2^L c_i + r_i && \text{(formula for } c_i\text{)}
\end{aligned}$$

This completes the induction and proves the claim for each i from 1 to k. We know that $c_k = a$ and $\prod b_i = b$, hence substituting $i \to k$ in the claim gives $b \otimes r_0 = (a, r_k)$. But this is just the full-width product of b with a random word, so we can apply Lemire's criterion [27]: if r_0 is an L-bit random word such that $r_k \geq (2^L \bmod b)$ then a is a uniformly

random integer in $[0, b)$.

By Lemma 1, since a is uniformly random in $[0, b)$, its mixed-radix digits in base $(b_1, b_2, \ldots, b_k)$ are independent and uniformly random in the ranges $[0, b_i)$. But those digits are $(a_1, a_2, \ldots, a_k)$, so the theorem is proved. Each a_i produced this way is a uniformly random integer in $[0, b_i)$, and the a_i are independent, provided that $r_k \geq (2^L \bmod b)$ and r_0 was picked uniformly at random.

5 | IMPLEMENTATION

In some applications the values of b_i are known ahead of time, possibly even at compile time. In that case the value of $t = (2^L \bmod b)$ can be precomputed, and Theorem 1 can be implemented succinctly as shown in Algorithm 2. In other applications the values of b_i are not known ahead of time. In that case the threshold t must be computed when needed, which involves a division operation. It can be avoided when $r_k \geq b$, as shown in Algorithm 3.

Algorithm 2 — Batched dice rolls (known threshold)

Require: Source of uniformly random integers in $[0, 2^L)$
Require: Target intervals $[0, b_i)$ for i in $1 \ldots k$, with $1 \leq \prod b_i \leq 2^L$
Require: The value $t = (2^L \bmod \prod b_i)$
Ensure: The a_i are independent and uniformly random in $[0, b_i)$

1: **repeat**
2: $\quad r \leftarrow$ random integer in $[0, 2^L)$
3: $\quad$ **for** i in $1 \ldots k$ **do**
4: $\quad\quad (a_i, r) \leftarrow b_i \otimes r$ ▷ Full-width multiply
5: **until** $r \geq t$
6: **return** $(a_1, a_2, \ldots, a_k)$

We must have $b \leq 2^L$ in order to use Theorem 1, and for Algorithm 3, we would prefer to have b at least an order of magnitude smaller than 2^L. If b is too close to 2^L then there is a high probability of taking the slow path that needs to calculate t, and possibly having to reroll the whole batch of dice. In many applications it is possible to bound the b_i in such a way that a value u satisfying $b \leq u \ll 2^L$ is known ahead of time. This allows for a faster implementation that avoids computing b most of the time, by enclosing lines 4–10 of Algorithm 3 within an "if $r < u$" block. We review such an approach in § 6.

Whichever version of the algorithm is used, at its core is a loop containing a single full-width multiplication "$b_i \otimes r$". The value of b_i is known, but each pass through the loop computes the value of r that will be used for the next iteration. This constitutes a loop-carried dependency, and it means that each iteration must complete before the next can begin. Modern commodity processors generally have the ability to carry out more than one operation at a time, a feature called instruction-level parallelism or superscalarity [30]. We expect that the computation of Algorithm 3 can be executed while other operations are completed (e.g., memory loads and stores), or the processor might speculatively use generated values of a_i for upcoming operations if the probability of $r < b$ is low. It would also be possible to interleave more than one batch of dice rolls.

Algorithm 3 — Batched dice rolls (unknown threshold)

Require: Source of uniformly random integers in $[0, 2^L)$
Require: Target intervals $[0, b_i)$ for i in $1 \dots k$, with $1 \le \prod b_i \le 2^L$
Ensure: The a_i are independent and uniformly random in $[0, b_i)$

```
r ← random integer in [0, 2^L)
for i in 1 ... k do
    (a_i, r) ← b_i ⊗ r                          ▷ Full-width multiply
b ← ∏ b_i
if r < b then
    t ← (2^L mod b)
    while r < t do
        r ← random integer in [0, 2^L)
        for i in 1 ... k do
            (a_i, r) ← b_i ⊗ r
return (a_1, a_2, ..., a_k)
```

6 | SHUFFLING ARRAYS

The Fisher-Yates shuffle of Algorithm 1 is widely used for permuting the elements of an array. It requires $n-1$ dice rolls to shuffle n elements. With a traditional implementation this involves $n-1$ calls to a random number generator, but by rolling dice in batches of k we can reduce that by a factor of k. As written, Algorithm 3 requires the computation of the product $b = \prod b_i$. In a batched shuffle, that is $b = n^{\underline{k}}$. One key insight is that we can replace an exact computation by an upper bound $u \ge b$. As long as $r_k \ge u$, there is no need to compute b exactly.

We denote by n_k the largest array length n at which we will use batches of k dice. We want an upper bound $u \ge n^{\underline{k}}$, ideally with $u \ll 2^L$ so the fast path succeeds with high probability. One possible choice is $u_k = n_k{}^{\underline{k}}$, which is the largest product we will ever see for a batch of k dice. We could use u_k as the upper bound for all batches of size k, however to improve efficiency we would like to lower the value of u as n decreases. A convenient approach is, whenever the current upper bound fails and we need to compute the true product, we assign that product to u and use it for subsequent batches of size k in the shuffle.

We illustrate this approach in Algorithm 4, which carries out the dice rolls and swaps for a single batch of size k, when there are n elements to shuffle. It takes an upper bound u as input, and at the end returns an upper bound for the next iteration. Usually the return value equals the input, however if the batch needed to calculate its true product then the return value equals that product.

To shuffle a full array, we first select the largest k such that $n_k \ge n$, and set $u = u_k$. Then we shuffle in batches of k, updating u along the way, until $n \le n_{k+1}$. At that point we set $u = u_{k+1}$ and shuffle in batches of $k+1$, and so forth up to some predetermined maximum batch size. Finally, when the number of remaining elements becomes smaller than the previous batch size, we finish the shuffle with one last batch.

Algorithm 4 — Batched partial shuffle (immediate swap)

Require: Source of uniformly random integers in $[0, 2^L)$
Require: Array z whose first n elements need to be shuffled
Require: Batch size $k \leq n$ for which $n^{\underline{k}} \leq 2^L$
Require: Upper bound $u \geq n^{\underline{k}}$
Ensure: Only the first $(n - k)$ elements of z remain to be shuffled

```
r ← random integer in [0, 2^L)
for i in 1 ... k do
    (a, r) ← (n + 1 − i) ⊗ r                ▷ Full-width multiply
    exchange z[a] and z[n−i]                ▷ Zero-based indexing
if r < u then
    u ← n^(k)                               ▷ Falling factorial
    t ← (2^L mod u)
    while r < t do
        r ← random integer in [0, 2^L)
        for i in 1 ... k do
            (a, r) ← (n + 1 − i) ⊗ r
            exchange z[a] and z[n−i]
return u                                    ▷ For the next batch
```

6.1 | Batch sizes

The batch size k presents a tradeoff. On one hand, we want to roll as many dice as we can with each random word. On the other hand, we want each batch to succeed on the first try with high probability so we do not have to reroll. These goals are in opposition, and we seek a balance between them. The question becomes, at what array length n should we start rolling dice in batches of k. In other words, what values of n_k should be used. This depends on the bit-width L, and can only truly be answered through benchmarks on the target hardware. However, a preliminary analysis can help to identify the right ballpark.

One possibility is to maximize the expected number of dice which succeed on the first try. That would suggest choosing n_k so the product $n_k{}^{\underline{k}}$ is about $2^L/k$, where the expected number of successful dice rolls is approximately $k-1$ for either a batch of size k or $k-1$. However this is not optimal, because the second roll is much more computationally expensive than the first, as it incurs the cost of calculating both $n^{\underline{k}}$ and $(2^L \bmod n^{\underline{k}})$.

Instead, we can estimate the cost for each die roll. Let us assume that calling the random number generator is as fast as 2 multiplications, and dividing is as slow as 16 multiplications. These are conservative estimates, since a fast random number generator reduces the benefit of batching, and a slow division operation increases the cost of rerolling. Using those parameters, and assuming that $u \approx n^{\underline{k}}$, we can find the cost per element for batches of size k at each n. By choosing n_k so this cost is cheaper with a batch of k than $k-1$ for all n up to n_k, but not $n_k + 1$, we obtain the values shown in Table 1.

These are likely to be overestimates for the optimal n_k because we have omitted some of the cost (e.g., misprediction cost). As a rough attempt to adjust for this, we may reduce the values of n_k from Table 1 to the next-lower power of 2. In the 64-bit case, we propose the following bounds:

- $n_2 = 2^{30}$ or 1 073 741 824

TABLE 1 Estimated values for n_k

k	$L = 64$	$L = 32$
2	1 358 187 913	20 724
3	929 104	581
4	26 573	109
5	3225	
6	815	
7	305	
8	146	

- $n_3 = 2^{19}$ or 524 288
- $n_4 = 2^{14}$ or 16 384
- $n_5 = 2^{11}$ or 2048
- $n_6 = 2^9$ or 512

7 | EXPERIMENTS

We have implemented Algorithm 4 in the C programming language, and used it for a batched Fisher-Yates shuffle. To help ensure that our results are reproducible, we make our source code freely available.[1] We use three different random number generators:

- Our fastest generator is a linear congruential generator proposed by Lehmer. It has good statistical properties [2]. Using a 128-bit integer seed as the state, we multiply it by the fixed 64-bit integer `0xda942042e4dd58b5`. The most-significant 64 bits of the resulting state are returned.
- Our second generator is a 64-bit version of O'Neill's PCG [7]. This relies on a 128-bit parameter m acting as a multiplier (`0x2360ed051fc65da44385df649fccf645`). With each call, a 128-bit state variable s is updated: $s \leftarrow ms + c$ where s and c are initialized once. The 64-bit random value is generated from the 128-bit state with a bit rotate and an exclusive or. We use O'Neill's own implementation adapted to our code base.
- Finally, we use ChaCha as a 64-bit cryptographically strong generator [31]. The chosen implementation[2] is written in conventional C, without advanced optimizations.

We measure the speed of this shuffle across a range of array lengths from 2^{13} through 2^{18} elements (8192 through 262 144), with a maximum batch size of either 2 or 6. Each element of the array occupies 64 bits and the entire arrays fit in CPU cache. To get accurate measurements, we shuffle the same array until the total elapsed time is 100 µs, and we record both the average and the minimum time per shuffle. We use the difference between the average and the minimum as an indication of our precision.

We provide the C source code for shuffling with batch sizes up to 6 in Appendix A. We use two different computer architectures, Apple M2 and Intel Ice Lake, see Table 2. We compare the results to a standard Fisher-Yates shuffle without any batching, but using an efficient bounded-number algorithm [27].

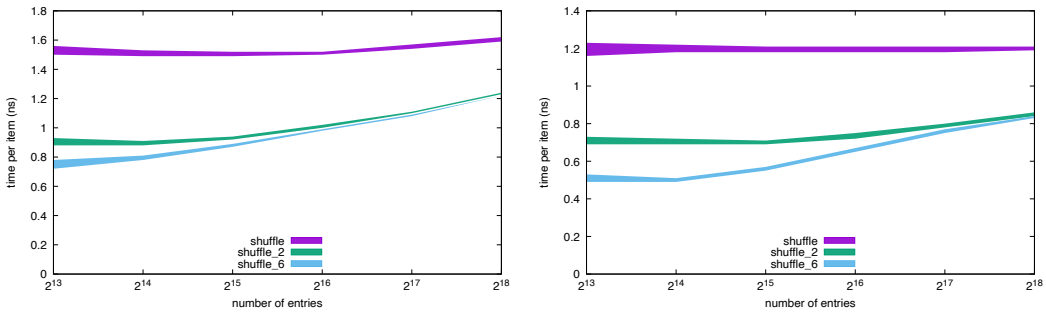
[1] `https://github.com/lemire/batched_random`

[2] `https://github.com/nixberg`

TABLE 2 Systems

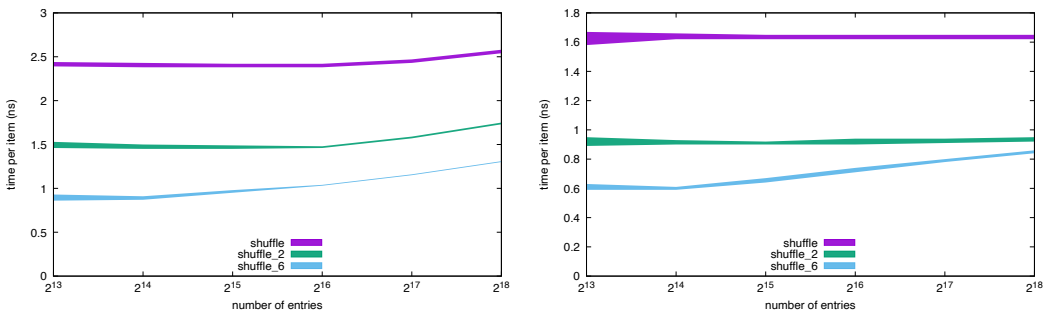
Processor	Intel Xeon Gold 6338	Apple M2
Frequency	2.0 GHz to 3.2 GHz	up to 3.49 GHz
Microarchitecture	Ice Lake (x64, 2019)	Avalanche (aarch64, 2022)
Memory	DDR4 (3200 MT/s)	LPDDR5 (6400 MT/s)
Compiler	GCC 12	Apple/LLVM 14
Cache (LLC)	48 MiB	16 MiB

For each architecture and random number generator, we plot the resulting time per element to shuffle an array using each maximum batch size. See Figure 1, Figure 2 and Figure 3. Normalizing per element keeps the scale of the result stable, and makes the relative speeds easy to see. In these graphs, "shuffle" is the standard unbatched Fisher-Yates shuffle, "shuffle_2" uses batches of 2, and "shuffle_6" uses batches of up to 6.



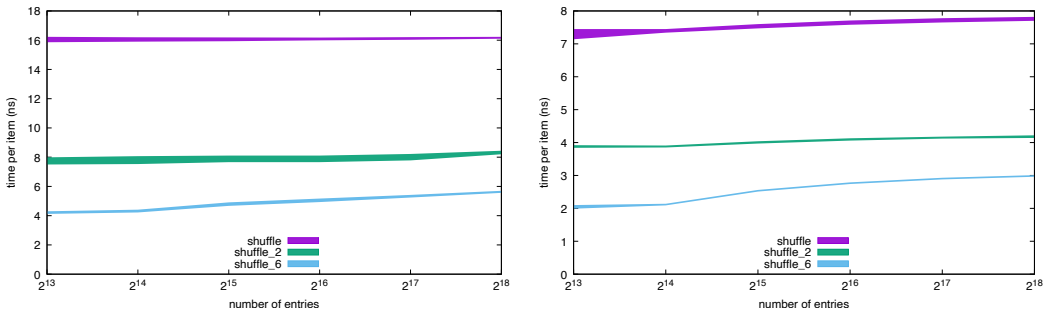
(a) Intel Ice Lake (b) Apple M2

FIGURE 1 Shuffle timings with Lehmer random number generator



(a) Intel Ice Lake (b) Apple M2

FIGURE 2 Shuffle timings with PCG random number generator



(a) Intel Ice Lake (b) Apple M2

FIGURE 3 Shuffle timings with ChaCha random number generator

The thickness of each curve shows the range between the fastest run and the average run in our tests. Across all the graphs, we see a similar trend. Unbatched shuffles are the slowest, batches of 2 are in the middle, and batches of up to 6 are the fastest. Note that for the range of sizes shown in the graphs, shuffle_6 uses batches of 3 above length 2^{14} where it switches to batches of 4, then to 5 at 2^{11} and 6 at 2^{9}. It would use batches of 2 from 2^{30} until 2^{19}, but that is outside the graphed region.

With the ChaCha random number generator, the speedups approach the theoretical ideal of $2\times$ for batches of 2, $3\times$ for batches of 3, and $4\times$ for batches of 4. The faster generators show proportionally smaller speedups, especially the Lehmer random number generator at large array sizes where the benefit of batching is only about $1.3\times$

To better understand these results, we use performance counters to record the number of instructions retired by our functions during the shuffling. The number of instructions does not, by itself, determine the performance because a variety of factors affect the number of instructions that can be executed in a given unit of time. In particular, cache and memory latency limit our maximal speed. Nevertheless, we find that our batched procedures (shuffle_2 and shuffle_6) use significantly fewer instructions per element than a conventional unbatched shuffle. See Table 3 where we consider the case when there are 16 384 elements in the array to be shuffled.

The reduction in the number of instructions between a conventional shuffle function and our most aggressively batched one (shuffle_6) is about 50% in the case of the Lehmer generator, and rises up to a threefold reduction when using the ChaCha generator. This suggests that the batched shuffles run at a higher speed in large part because they use far fewer instructions. In turn, they use fewer instructions because fewer calls to the random number generator are required.

TABLE 3 Instructions retired per element (16 384 elements)

generator	function	Intel Xeon Gold 6338	Apple M2
Lehmer	shuffle	2.2	2.2
	shuffle_2	1.6	1.8
	shuffle_6	1.1	1.4
PCG64	shuffle	3.3	3.0
	shuffle_2	2.3	2.2
	shuffle_6	1.4	1.6
ChaCha	shuffle	17.4	16.6
	shuffle_2	9.4	9.0
	shuffle_6	4.8	4.8

8 | CONCLUSION

We have shown that Lemire's nearly-divisionless method of generating bounded random integers can be extended to generate multiple such numbers from a single random word. When rolling several dice or shuffling an array, this batched approach can reduce the number of random bits used, without increasing the amount of computation required. Though our approach is more beneficial when the random bits are more computationally expensive, we still find it beneficial with a fast random number generator (e.g., Lehmer [2]).

Our results are based on a system-oblivious computational model described in § 6. We expect that further tuning, especially system-specific tuning, might have some additional benefits. Our implementation is in the C language. We expect that our good results should carry over to other languages such as C++, Rust, Go, Swift, Java, C# and so forth with relative ease. However, care might be needed to ensure that the generated compiled code is comparable to the result of our C code. We expect that our approach can find broad applications. Future work should examine other applications such as sampling algorithms and simulations, as well as the shuffling of very large and very small arrays.

A | CODE SAMPLES

Unbatched shuffle:

```
// generates a bounded random integer
uint64_t random_bounded(uint64_t range, uint64_t (*rng)(void)) {
  __uint128_t random64bit, multiresult;
  uint64_t leftover;
  uint64_t threshold;
  random64bit = rng();
  multiresult = random64bit * range;
  leftover = (uint64_t)multiresult;
  if (leftover < range) {
    threshold = -range % range;
    while (leftover < threshold) {
      random64bit = rng();
      multiresult = random64bit * range;
      leftover = (uint64_t)multiresult;
    }
  }
  return (uint64_t)(multiresult >> 64);
}
```

```
// Fisher-Yates shuffle
void shuffle(uint64_t *storage, uint64_t size,
             uint64_t (*rng)(void)) {
  uint64_t i;
  for (i = size; i > 1; i--) {
    uint64_t nextpos = random_bounded(i, rng);
    uint64_t tmp = storage[i - 1];
    uint64_t val = storage[nextpos];
    storage[i - 1] = val;
    storage[nextpos] = tmp;
  }
}
```

Batched shuffle:

```
// performs k steps of a shuffle
void batched(uint64_t *storage, uint64_t n, uint64_t k,
             uint64_t bound, uint64_t (*rng)(void)) {
  __uint128_t x;
  uint64_t r = rng();
  uint64_t pos1, pos2;
  uint64_t val1, val2;
  for (uint64_t i = 0; i < k; i++) {
    x = (__uint128_t)(n - i) * (__uint128_t)r;
    r = (uint64_t)x;
    pos1 = n - i - 1; pos2 = (uint64_t)(x >> 64);
    val1 = storage[pos1]; val2 = storage[pos2];
    storage[pos1] = val2; storage[pos2] = val1;
  }
  if (r < bound) {
    bound = n;
    for (uint64_t i = 1; i < k; i++) {
      bound *= n - i;
    }
    uint64_t t = -bound % bound;
    while (r < t) {
      r = rng();
      for (uint64_t i = 0; i < k; i++) {
        x = (__uint128_t)(n - i) * (__uint128_t)r;
        r = (uint64_t)x;
        pos1 = n - i - 1; pos2 = (uint64_t)(x >> 64);
        val1 = storage[pos1]; val2 = storage[pos2];
        storage[pos1] = val2; storage[pos2] = val1;
      }
    }
  }
  return bound;
}

// Fisher-Yates shuffle, rolling up to six dice at a time
void shuffle_6(uint64_t *storage, uint64_t size,
               uint64_t (*rng)(void)) {
  uint64_t i = size;
  for (; i > 1 << 30; i--) {
    batched(storage, i, 1, i, rng);
  }
  // Batches of 2 for sizes up to 2^30 elements
  uint64_t bound = (uint64_t)1 << 60;
  for (; i > 1 << 19; i -= 2) {
    bound = batched(storage, i, 2, bound, rng);
  }
  // Batches of 3 for sizes up to 2^19 elements
  bound = (uint64_t)1 << 57;
  for (; i > 1 << 14; i -= 3) {
    bound = batched(storage, i, 3, bound, rng);
  }
  // Batches of 4 for sizes up to 2^14 elements
  bound = (uint64_t)1 << 56;
  for (; i > 1 << 11; i -= 4) {
    bound = batched(storage, i, 4, bound, rng);
  }
  // Batches of 5 for sizes up to 2^11 elements
  bound = (uint64_t)1 << 55;
  for (; i > 1 << 9; i -= 5) {
    bound = batched(storage, i, 5, bound, rng);
  }
  // Batches of 6 for sizes up to 2^9 elements
  bound = (uint64_t)1 << 54;
  for (; i > 6; i -= 6) {
    bound = batched(storage, i, 6, bound, rng);
```

```
    }
    if (i > 1) {
      batched(storage, i, i - 1, 720, rng);
    }
  }
```

references

[1] Matsumoto M, Nishimura T. Mersenne Twister: A 623-dimensionally Equidistributed Uniform Pseudo-random Number Generator. ACM Trans Model Comput Simul 1998 Jan;8(1):3–30.

[2] L'Ecuyer P. Tables of linear congruential generators of different sizes and good lattice structure. Mathematics of Computation 1999;68(225):249–260.

[3] L'Ecuyer P, Blouin F, Couture R. A Search for Good Multiple Recursive Random Number Generators. ACM Trans Model Comput Simul 1993 Apr;3(2):87–98.

[4] De Matteis A, Pagnutti S. Parallelization of random number generators and long-range correlations. Numerische Mathematik 1988;53(5):595–608.

[5] Fishman G. Monte Carlo: concepts, algorithms, and applications. corrected edition ed. Berlin: Springer Science & Business Media; 2013.

[6] L'Ecuyer P. Random Numbers for Simulation. Commun ACM 1990 Oct;33(10):85–97.

[7] O'neill ME. PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. ACM Transactions on Mathematical Software 2014;.

[8] L'Ecuyer P. History of uniform random number generation. In: Proceedings of the 2017 Winter Simulation Conference New York, NY, USA: IEEE Press; 2015. p. 1–17.

[9] L'Ecuyer P. Random number generation. In: Handbook of Computational Statistics Berlin: Springer; 2012.p. 35–71.

[10] Sleem L, Couturier R. TestU01 and Practrand: Tools for a randomness evaluation for famous multimedia ciphers. Multimedia Tools and Applications 2020;79(33):24075–24088.

[11] Vitter JS. Random Sampling with a Reservoir. ACM Trans Math Softw 1985 Mar;11(1):37–57.

[12] Knuth DE. Seminumerical Algorithms, vol. 2 of The Art of Computer Programming. Boston: Addison-Wesley; 1969.

[13] Durstenfeld R. Algorithm 235: Random Permutation. Commun ACM 1964 Jul;7(7):420–.

[14] Durstenfeld R. Algorithm 235: random permutation. Communications of the ACM 1964;7(7):420.

[15] Devroye L. Random Variate Generation for Multivariate Unimodal Densities. ACM Trans Model Comput Simul 1997 Oct;7(4):447–477.

[16] Calvin JM, Nakayama MK. Using Permutations in Regenerative Simulations to Reduce Variance. ACM Trans Model Comput Simul 1998 Apr;8(2):153–193.

[17] Owen AB. Latin Supercube Sampling for Very High-dimensional Simulations. ACM Trans Model Comput Simul 1998 Jan;8(1):71–102.

[18] Osogami T. Finding Probably Best Systems Quickly via Simulations. ACM Trans Model Comput Simul 2009 Aug;19(3):12:1–12:19.

[19] Amrein M, Künsch HR. A Variant of Importance Splitting for Rare Event Estimation: Fixed Number of Successes. ACM Trans Model Comput Simul 2011 Feb;21(2):13:1–13:20.

[20] Hernandez AS, Lucas TW, Carlyle M. Constructing Nearly Orthogonal Latin Hypercubes for Any Nonsaturated Run-variable Combination. ACM Trans Model Comput Simul 2012 Nov;22(4):20:1–20:17.

[21] Hinrichs C, Ithapu VK, Sun Q, Johnson SC, Singh V. Speeding Up Permutation Testing in Neuroimaging. In: Proceedings of the 26th International Conference on Neural Information Processing Systems NIPS'13, USA: Curran Associates Inc.; 2013. p. 890–898.

[22] Shterev ID, Jung SH, George SL, Owzar K. permGPU: Using graphics processing units in RNA microarray association studies. BMC Bioinformatics 2010 Jun;11(1):329. https://doi.org/10.1186/1471-2105-11-329.

[23] Langr D, Tvrdík P, Dytrych T, Draayer JP. Algorithm 947: Paraperm—Parallel Generation of Random Permutations with MPI. ACM Trans Math Softw 2014 Oct;41(1):5:1–5:26.

[24] Sanders P. Random permutations on distributed, external and hierarchical memory. Information Processing Letters 1998;67(6):305–309.

[25] Gustedt J. In: Engineering Parallel In-Place Random Generation of Integer Permutations Berlin, Heidelberg: Springer Berlin Heidelberg; 2008. p. 129–141.

[26] Waechter M, Hamacher K, Hoffgaard F, Widmer S, Goesele M. In: Is Your Permutation Algorithm Unbiased for $n \neq 2^m$? Berlin, Heidelberg: Springer Berlin Heidelberg; 2012. p. 297–306.

[27] Lemire D. Fast Random Integer Generation in an Interval. ACM Transactions on Modeling and Computer Simulation 2019 Jan;29(1):1–12. http://dx.doi.org/10.1145/3230636.

[28] Abel A, Reineke J. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems; 2019. p. 673–686.

[29] Von Neumann J. Various techniques used in connection with random digits. National Bureau of Standards Series 1951;12:36–38.

[30] Shen JP, Lipasti MH. Modern processor design: fundamentals of superscalar processors. Waveland Press; 2013.

[31] Bernstein DJ. ChaCha, a variant of Salsa20. In: Workshop record of SASC, vol. 8; 2008. p. 3–5.