

Optimized random number generation in an interval

Michael Orlov

Department of Computer Science, Ben-Gurion University of the Negev, PO Box 653, Beer-Sheva 84105, Israel

ARTICLE INFO

Article history:

Received 17 January 2008
Available online 14 March 2009
Communicated by F. Dehne

Keywords:

Random number generation
Algorithms

ABSTRACT

We present a universal optimization for generating a uniformly distributed integer in an interval, based on the underlying uniform distribution. This optimization provides up to 25% run-time improvement, and what is sometimes more important, up to 25% reduction in usage of (pseudo-)random bits. The optimization entails no run-time penalty for all but the most primitive pseudo-random number generators, and can be easily employed by existing software and hardware. For hardware implementations we present additional improvements.

© 2009 Elsevier B.V. All rights reserved.

1. Introduction

Many (pseudo-)random number generators (RNGs), both software and hardware based, build upon a basic component that produces chunks of (pseudo-)random bits, typically 32-bit words [1]. Each invocation of the basic component has either real or amortized time cost, which is expensive relative to the basic arithmetic operations—usually moderately expensive in software-based generators, and more expensive in hardware-based generators [2].

One of the most frequent RNG operations employed in software simulations, games, AI, etc., is the generation of uniformly distributed integers in a given interval $\mathcal{U}[0, \dots, m)$. Here, the desired number is one of $\{0, \dots, m-1\}$, whereas the RNG produces numbers in $\mathcal{U}[0, \dots, M)$, where $m \leq M$, and M is typically a power of 2.

The straightforward approach of taking $r \leftarrow \mathcal{U}[0, \dots, M)$ modulo m introduces an undesirable skew of probabilities, if m is not small relative to M and a quality RNG is used. For instance, when taking $m = \frac{2}{3}M$ and employing this approach, values in $[0, \dots, \frac{m}{2})$ have twice the probability of values in $[\frac{m}{2}, \dots, m)$.

```

1  $r \leftarrow M \bmod m$ 
2 repeat
3    $u \leftarrow \mathcal{U}[0, \dots, M)$ 
4 until  $u < M - r$ 
5 return  $u \bmod m$ 

```

Fig. 1. RANDOM-STANDARD(m): Standard algorithm for getting a uniform distribution on $\mathcal{U}[0, \dots, m)$ from $\mathcal{U}[0, \dots, M)$, where $m \leq M$.

The regular technique used to solve this problem is to reject the “remainder” values when they are returned by RNG. This technique is shown in Fig. 1.¹

However, in this way the entropy of the remainder values is discarded. We show further in this paper that these values can be reused in order to reduce the probability of rejection during the next call to produce a value in $\mathcal{U}[0, \dots, M)$. The reduction in the expected number of the calls to RNG can reach 25%.

The idea and the algorithm for the reuse of remainder values is presented in Section 2. Afterwards, the GCD function in the algorithm is replaced with bitwise operations, so that the algorithm benefits even fast RNGs such as Mersenne Twister (Section 3). In Section 4, the algorithm is optimized for specialized hardware, which can have cus-

¹ The algorithms we present are optimized for clarity. In actual implementation some tricks are necessary, since in most cases M is not expressible as a single architecture word. Thus, $M' = M - 1$ needs to be used, and, e.g., instead of $M \bmod m$, $(M' - (m - 1)) \bmod m$ can be computed. Also, compiler hints for likely and unlikely branches, and other optimizations, improve performance.

E-mail address: orlov@m@cs.bgu.ac.il.

```

1  $r \leftarrow M \bmod m$ 
2  $g \leftarrow \gcd(m, r)$ 
3 if  $g = 1$  then
4   return RANDOM-STANDARD( $m$ )
5 else
6    $u \leftarrow \mathcal{U}[0, \dots, M]$ 
7   if  $u < M - r$  then
8     return  $u \bmod m$ 
9   else
10     $u \leftarrow u - (M - r)$ 
11     $m \leftarrow m \div g$ 
12    return  $m \cdot (u \bmod g) + \text{RANDOM-STANDARD}(m)$ 

```

Fig. 2. RANDOM-REUSE(m): Improved algorithm for uniform distribution on $\mathcal{U}[0, \dots, m]$, where invalid values in $\mathcal{U}[0, \dots, M]$ are reused.

tom bitwise instructions. That is, hardware RNGs can efficiently provide integer values in a given interval. Theoretical optimization statistics, and experimental results for a software-based RNG are presented in Section 5. The paper is concluded in Section 6.

2. Reuse of remainder values

In the rejection case in Fig. 1, the values returned by the RNG are uniformly distributed: $u \in \mathcal{U}[M - r, \dots, M]$, and therefore $u - (M - r) \in \mathcal{U}[0, \dots, r]$. The range $[0, \dots, m]$ can thus in principle be partitioned into r equal sub-intervals, and in the next iteration of the algorithm, m becomes $\frac{m}{r}$, whereas the original $u - (M - r)$ value denotes the sub-interval number—the uniform distribution properties are kept intact.

However, $[0, \dots, m]$ cannot always be partitioned into r equal intervals, and hence the algorithm shown in Fig. 2 can be executed, where there are $\gcd(m, r)$ instead of r sub-intervals.

It should be noted that the sub-interval call (line 12) is to RANDOM-STANDARD, and not a recursive call. This is because a reuse cannot happen twice:

$$\begin{aligned} \gcd\left(\frac{m}{g}, M \bmod \frac{m}{g}\right) &= \gcd\left(M, \frac{m}{g}\right) \\ &= \gcd\left(M, \frac{m}{\gcd(M, m)}\right) = 1. \end{aligned} \quad (1)$$

3. Optimizing reuse of values

Greatest common divisor computation in Fig. 2 can be quite expensive, to the point that using it with fast pseudo-random number generators such as Mersenne Twister is grossly inefficient, as shown in Section 5. This is unfortunate, because instead of being a universal algorithm, which can be used anywhere, the scope of RANDOM-REUSE is limited to “expensive” RNGs, such as hardware and cryptography-oriented ones.

One way to approach this problem is to construct dispatch branches in advance, either in run-time with generator functions, or in compile-time when the language possesses the necessary capabilities [3].

However, it is nearly universal that M is a power of 2—typically, $M = 2^{32}$. In (1), the equivalence $g = \gcd(M, m)$ has been noted. For $M = 2^k$, it is easy to see that $g = 2^{k'}$, where k' is the number of trailing zeros in the binary representation of m .

```

1  $g \leftarrow m \wedge -m$ 
2 if  $g = 1$  then
3   return RANDOM-STANDARD( $m$ )
4 else
5    $r \leftarrow M \bmod m$ 
6    $u \leftarrow \mathcal{U}[0, \dots, M]$ 
7   if  $u < M - r$  then
8     return  $u \bmod m$ 
9   else
10     $u \leftarrow u - (M - r)$ 
11     $m \leftarrow m \div g$ 
12    return  $m \cdot (u \wedge (g - 1)) + \text{RANDOM-STANDARD}(m)$ 

```

Fig. 3. RANDOM-FAST(m): Improved algorithm for uniform distribution on $\mathcal{U}[0, \dots, m]$, where invalid values in $\mathcal{U}[0, \dots, M]$ are reused, and $M = 2^k$ for some k .

On a general-purpose architecture, in the absence of an instruction for counting trailing zeros,² g can be computed without loops as follows:

$$\begin{aligned} m &= x \dots xx1 \overbrace{0 \dots 00}^{n \geq 0} \\ -m &= \bar{x} \dots \bar{x}\bar{x}10 \dots 00 \\ g = m \wedge -m &= 0 \dots 0010 \dots 00. \end{aligned}$$

Application of this method is shown in Fig. 3, where a modulo operation is also replaced with a bitwise one (line 12).

This algorithm can be used for running time optimization anywhere with a non-trivial RNG instead of the usual remainder values rejection approach. With trivial (i.e., linear congruential) generators handling remainder values at all does not make sense, since the RNG quality is not high enough—two or more calls to the RNG can be made instead, effectively using M^2 limit instead of M .

4. Optimization on specialized hardware

When specialized hardware is employed, for instance, a PCI RNG card, its driver can provide an API for generating a (pseudo-)random number in $\mathcal{U}[0, \dots, m]$. In this case, a trailing-bits counting instruction can be efficiently implemented directly, and the algorithm shown in Fig. 4 can be hardwired.

5. Evaluation

The theoretical optimization limit is an improvement of 25%: going down from the expected 2 calls to $\mathcal{U}[0, \dots, M]$ per each call to $\mathcal{U}[0, \dots, m]$ when the probability of rejection is $\frac{1}{2}$, to 1.5 expected calls to $\mathcal{U}[0, \dots, M]$ when the probability of first rejection is $\frac{1}{2}$, and after the reduction of m the probability of another rejection is 0.

Table 1 shows execution statistics for a specific Mersenne Twister implementation [4], when taking $m = 2^{31} + 32$ on 32-bit platform.

Whereas there is indeed an improvement of 24.60% in terms of number of calls to the underlying random number generator for each call to RANDOM-*, the improvement

² Or an instruction which can be used to that effect, such as POPCNT in the SSE 4.2 instruction set for the Intel Core architecture.

```

1 log g ← NTL(m)
2 if log g ≠ 0 then
3   r ← M mod m
4   u ← U[0, ..., M)
5   if u < M - r then
6     return u mod m
7   else
8     u ← u - (M - r)
9     m ← m ≫ log g
10  return m · (u ∧ ((1 ≪ log g) - 1)) + RANDOM-STANDARD(m)
11 else
12  return RANDOM-STANDARD(m)

```

Fig. 4. RANDOM-FPGA(m): Improved algorithm for uniform distribution on $\mathcal{U}[0, \dots, m)$, where invalid values in $\mathcal{U}[0, \dots, M)$ are reused, $M = 2^k$ for some k , and the NTL (number of trailing zeros) instruction is available. “ \ll ” and “ \gg ” denote, respectively, shift-left and shift-right instructions.

Table 1

Comparing RANDOM- * on $m = 2^{31} + 32$, with 50,000,000 iterations. Randomness source: Mersenne Twister 19937 (same seed).

Algorithm	Average calls to $\mathcal{U}[0, \dots, M)$	Time
RANDOM-STANDARD	1.99996	1.50 s
RANDOM-REUSE	1.50789	6.59 s
RANDOM-FAST	1.50789	1.46 s

Platform: Pentium-IV, 2.8 GHz, Intel compiler 10.1 with IPO enabled.

Table 2

Comparing RANDOM- * on $m = 2^{31} + 32$, with 1,000,000 iterations. Randomness source: Linux 2.6.22, /dev/urandom.

Algorithm	Average calls to $\mathcal{U}[0, \dots, M)$	Time
RANDOM-STANDARD	1.99949	2.23 s
RANDOM-REUSE	1.50813	1.79 s
RANDOM-FAST	1.50783	1.68 s

Platform: Pentium-IV, 2.8 GHz, Intel compiler 10.1 with IPO enabled.

in the run-time for RANDOM-FAST is only 2.67%. This is to be expected with a fast generator, where amortized time spent on each random value production is comparable with the time spent on auxiliary computations in RANDOM-FAST. Of note here, however, is that there is no run-time penalty for the additional computations (as happens with RANDOM-REUSE), and that the algorithm in Fig. 3 is universally applicable.

When a security-oriented RNG is employed (software or hardware), the number of calls to the underlying generator dominates the additional operations in RANDOM-FAST, as shown in Table 2 for a security-conscious pseudo-randomness source on a Linux system.

With RANDOM-STANDARD, the expected number of calls to $\mathcal{U}[0, \dots, M)$ per a single $\mathcal{U}[0, \dots, m)$ is given by

$$\frac{1}{\Pr[\text{success}]} = \frac{1}{1 - \Pr[\text{rejection}]} = \frac{1}{1 - \frac{r}{M}} = \frac{M}{M - r}.$$

When RANDOM-FAST is employed, and the random value is rejected, the sub-call with $\frac{m}{g}$ is to RANDOM-STANDARD, therefore the expected number of calls to $\mathcal{U}[0, \dots, M)$ in this case is

$$1 \cdot \left(1 - \frac{r}{M}\right) + \left(1 + \frac{1}{1 - \frac{r'}{M}}\right) \cdot \frac{r}{M} = 1 + \frac{r}{M - r'},$$

where $r' = M \bmod \frac{m}{g}$.

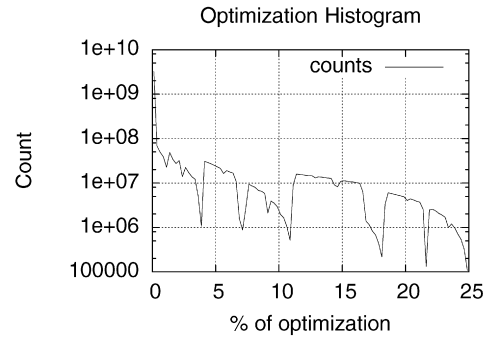


Fig. 5. Expected optimization histogram of values in $[0, \dots, M)$ for $M = 2^{32}$. Each point represents a $\frac{1}{4}$ %-wide bucket.

Thus, the expected run-time savings when using RANDOM-FAST instead of RANDOM-STANDARD, and when m reduces ($g > 1$), expressed in number of calls to $\mathcal{U}[0, \dots, M)$, are given by

$$\left[1 - \left(1 + \frac{r}{M - r'}\right) \cdot \frac{M - r}{M}\right] \cdot 100\%. \quad (2)$$

Fig. 5 shows a histogram of expected improvements in run-time for the most common M of 2^{32} , whereas all possible values for m are considered.

Taking the example in Table 1, where

$$\begin{aligned} M &= 2^{32}, & r &= 2^{31} - 32, \\ m &= 2^{31} + 32, & \frac{m}{g} &= 2^{26} + 1, \\ g &= 32, & r' &= 2^{26} - 63 \end{aligned}$$

the theoretical optimization given by (2) is $\approx 24.60\%$, which is equal to the experimental result in Table 1.

6. Conclusion

The optimization described here can be plugged as is in most random number generation APIs, to instantly provide up to 25% improvement in execution time and random bits consumption. Additionally, hardware implementations can employ techniques shown in Fig. 4 in order to gain additional runtime enhancements.

Acknowledgements

The author would like to thank M. Sipper and M. Goldberg for their helpful comments.

Michael Orlov is supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities, and is partially supported by the Lynn and William Frankel Center for Computer Sciences.

References

- [1] M. Matsumoto, T. Nishimura, Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, ACM Transactions on Modeling and Computer Simulation 8 (1) (1998) 3–30, doi:10.1145/272991.272995.

- [2] R.P. Brent, Fast and reliable random number generators for scientific computing, in: J. Dongarra, K. Madsen, J. Wasniewski (Eds.), *Applied Parallel Computing: State of the Art in Scientific Computing*, 7th International Conference, PARA 2004, Lyngby, Denmark, June 20–23, 2004, in: *Revised Selected Papers, Lecture Notes in Computer Science*, vol. 3732, Springer, Berlin, Heidelberg, 2006, pp. 1–10.
- [3] M. Orlov, Random numbers in a range using generic programming, *Dr. Dobbs' Journal* 33 (4) (2008) 44–48, <http://www.ddj.com/cpp/206904716>.
- [4] J. Bedaux, C++ Mersenne Twister pseudo-random number generator, C++ Library, retrieved on 2007-11-15 (Jan. 2003), <http://bedaux.net/mtrand/>.