Bit recycling for scaling random number generators

Andrea C. G. Mennucci[*]

December 21, 2010

Abstract

Many Random Number Generators (RNG) are available nowadays; they are divided in two categories, *hardware RNG*, that provide "true" random numbers, and *algorithmic RNG*, that generate pseudo random numbers (PRNG). Both types usually generate random numbers $(X_n)_n$ as independent uniform samples in a range $0, \dots 2^b - 1$, with $b = 8, 16, 32$ or $b = 64$. In applications, it is instead sometimes desirable to draw random numbers as independent uniform samples $(Y_n)_n$ in a range $1, \dots M$, where moreover M may change between drawings. Transforming the sequence $(X_n)_n$ to $(Y_n)_n$ is sometimes known as *scaling*. We discuss different methods for scaling the RNG, both in term of mathematical efficiency and of computational speed.

Contents

[*]Scuola Normale Superiore, Pisa, Italy (`a.mennucci@sns.it`)

1 Introduction

We consider the following problem. We want to generate a sequence of random numbers $(Y_n)_n$ with a specified probability distribution, using as input a sequence of random numbers $(X_n)_n$ uniformly distributed in a given range. There are various methods available; these methods involve transforming the input in some way; for this reason, these methods work equally well in transforming both pseudo-random and true random numbers. One such method, called the acceptance-rejection method, involves designing a specific algorithm, that pulls random numbers, transforms them using a specific function, tests whether the result satisfies a condition: if it is, the value is accepted; otherwise, the value is rejected and the algorithm tries again.

This kind of method has a defect, though: if not carefully implemented, it throws away many inputs. Let's see a concrete example. We suppose that we are given a RNG that produces random bits, evenly distributed, and independent[1]. We want to produce a random number R in the range $\{1, 2, 3\}$, uniformly distributed and independent. Consider the following method.

Example 1 *We draw two random bits; if the sequence is* 11*, we throw it away and draw two bits again; otherwise we return the sequence as* R*, mapping* $00, 01, 10$ *to* $1, 2, 3$*.*

This is rather wasteful! The entropy in the returned random R is $\log_2(3) = 1.585$bits; there is a $1/4$ probability that we throw away the input, so the expected number of (pair of tosses) is $4/3$ and then expected number of input bits is $8/3$; all together we are effectively using only

$$\frac{\log_2(3)}{8/3} = 59\%$$

of the input.

The waste may be consider as an unnecessary slowdown of the RNG: if the RNG can generate 1 bit in $1\mu s$, then, after the example procedure, the rate has decreased to $1.6\mu s$ per bit. Since a lot of effort was put in designing fast RNG in the near past, then slowing down the rate by $+60\%$ is simply unacceptable.

Another problem in the example method above is that, although it is quite unlikely, we can have a very long run of "00" bits. This means that we cannot guarantee that the above procedure will generate the next number in a predetermined amount of time.

There are of course better solutions, as this *ad hoc method.*

Example 2 ([1]) *We draw eight random bits, and consider them as a number* x *in the range* $0 \dots 255$*; if the number is more than* $3^5 - 1 = 242$*, we throw it away and draw eight bits again; otherwise we write* x *as five digits in base 3 and return these digits as 5 random samples.*

This is much more efficient! The entropy in the returned five samples is $5 \log_2(3) = 7.92$bits; there is a $13/256$ probability that we throw away the input, so the expected number of 8-tuples of inputs is $256/253$ and then expected number of input bits is $2048/253$; all together we are now using

$$\frac{5 \log_2(3)}{2048/253} = 97\%$$

of the input.

In this paper we will provide a mathematical proof (section 2), and discuss some method (section 3), to optimize the scaling of a RNG. Unfortunately after writing this paper it turned out that one of the ideas we are presenting in section 3, was already described in [1]. This paper contains the mathematical proof of the method, the discussion of how best to choose parameters, discussion of its efficiency, and numerical speed tests.

Remark 3 *A different approach may be to use a decompressing algorithm. Indeed, e.g., the* arithmetic encoder *decoding algorithm, can be rewritten to decode a stream of bits to an output of symbols with prescribed probability distributions* [2]*. Unfortunately, it is quite difficult to mathematically prove that such an approach really does transform a stream of independent equidistributed bits into an output of independent random variables. Also, the* arithmetic encoder *is complex, and this complexity would slow down the RNG, defeating one of the goals. (Moreover, the* arithmetic encoder *was originally heavily patented.)*

[1]For example, repeatedly tossing a coin with the faces labeled $0, 1$.

[2]If interested, I have the code somewhere in the closet

A note on notations. In all of the paper, *ns* is a *nanosecond*, that is 10^{-9}seconds. When x is a real number, $\lfloor x \rfloor = \texttt{floor}(x)$ is the largest integer that is less or equal than x.

2 Process splitting

Let $\mathbb{N} = \{0, 1, 2, 3, 4, 5 \ldots\}$ be the set of natural numbers.

Let $(\Omega, \mathcal{A}, \mathbb{P})$ a probability space, let $(E, \mathcal{E})$ be a measurable space, and $\overline{X}$ a process of i.i.d. random variables $(X_n)_{n\in\mathbb{N}}$ defined on $(\Omega, \mathcal{A}, \mathbb{P})$ and each taking values in $(E, \mathcal{E})$. We fix an event $S \in \mathcal{E}$ such that $\mathbb{P}\{X_i \in S\} \neq 0, 1$; we define $p_S \overset{\text{def}}{=} \mathbb{P}\{X_i \in S\}$.

We define a formal method of process splitting/unsplitting.

The **splitting** of $\overline{X}$ is the operation that generates three processes $\overline{B}, \overline{Y}, \overline{Z}$, where $\overline{B} = (B_n)_{n\in\mathbb{N}}$ is an i.i.d. Bernoulli process with parameter p_S, and $\overline{Y} = (Y_n)_{n\in\mathbb{N}}$ and $\overline{Z} = (Z_n)_{n\in\mathbb{N}}$ are processes taking values respectively in S and $E \setminus S$. The **unsplitting** is the opposite operation. These operations can be algorithmically and intuitively described by the following pseudocode (where processes are thought of as *queues of random variables*).

```
procedure SPLITTING($\overline{X} \mapsto (\overline{B}, \overline{Y}, \overline{Z})$)
    initialize the three empty queues $\overline{B}, \overline{Y}, \overline{Z}$
    repeat
        pop X from $\overline{X}$
        if $X \in S$ then
            push 1 onto $\overline{B}$
            push X onto $\overline{Y}$
        else
            push 0 onto $\overline{B}$
            push X onto $\overline{Z}$
        end if
    until forever
end procedure
```

```
procedure UNSPLITTING($(\overline{B}, \overline{Y}, \overline{Z}) \mapsto \overline{X}$)
    initialize the empty queue $\overline{X}$
    repeat
        pop B from $\overline{B}$
        if $B = 1$ then
            pop Y from $\overline{Y}$
            push Y onto $\overline{X}$
        else
            pop Z from $\overline{Z}$
            push Z onto $\overline{X}$
        end if
    until forever
end procedure
```

The fact that the *splitting* operation is invertible implies that no entropy is lost when splitting. We will next show a very important property, namely, that the splitting operation preserve probabilistic independence.

2.1 Mathematical formulation

We now rewrite the above idea in a purely mathematical formulation.

We define the Bernoulli process $(B_n)_{n\in\mathbb{N}}$ by

$$B_n \overset{\text{def}}{=} \begin{cases} 1 & X_n \in S \\ 0 & X_n \notin S \end{cases} \tag{1}$$

and the **times of return to success** as

$$U_0 = \inf\{k : k \geq 0, B_k = 1\} \tag{2}$$
$$U_n = \inf\{k : k \geq 1 + U_{n-1}, B_k = 1\}, \quad n \geq 1 \tag{3}$$

whereas the **times of return to unsuccess** are

$$V_0 = \inf\{k : k \geq 0, B_k = 0\} \tag{4}$$
$$V_n = \inf\{k : k \geq 1 + U_{n-1}, B_k = 0\}, \quad n \geq 1 \tag{5}$$

it is well known that $(U_n), (V_n)$ are (almost certainly) well defined and finite.

We eventually define the processes $(Y_n)_{n\in\mathbb{N}}$ and $(Z_n)_{n\in\mathbb{N}}$ by

$$Y_n = X_{U_n} \quad Z_n = X_{V_n} \tag{6}$$

Theorem 4 *Assume that $\overline{X}$ is a process of i.i.d. random variables. Let μ be the law of X_1. Then*

- *the random variables B_n, Y_n, Z_n are independent; and*
- *the variables of the same type are identically distributed: the variables B_n have parameter $\mathbb{P}\{B_n = 1\} = p_S$; the variables Y_n have law $\mu(\cdot \mid S)$; the variables Z_n have law $\mu(\cdot \mid E \setminus S)$.*

Proof. It is obvious that $\overline{B}$ is a Bernoulli process of independent variables with parameter $\mathbb{P}\{B_n = 1\} = p_S$.

Let $K, M \geq 1$ integers. Let $u_0 < u_1 < \dots u_K$ and $v_0 < v_1 < \dots v_M$ be integers, and consider the event

$$A \stackrel{\text{def}}{=} \{U_0 = u_0, \dots U_K = u_K, V_0 = v_0, \dots V_M = v_M\} \tag{7}$$

If $A \neq \emptyset$ then

$$A = \{B_0 = b_0, \dots, B_N = b_N\} \tag{8}$$

where $N = \max\{u_K, v_M\}$ and $b_j \in \{0, 1\}$ are suitably chosen. Indeed, supposing that $N = u_K > v_M$, then we use the success times, and set that $b_j = 1$ iff $j = u_k$ for a $k \leq K$; whereas supposing that $N = v_M > u_K$, then we use the unsuccess times, and set that $b_j = 0$ iff $j = v_m$ for a $m \leq M$.

Let $\mathcal{F}_{K,M}$ be the family of all above events A defined as per equation (7), for different choices of $(u_i), (v_j)$; let

$$\mathcal{F} = \bigcup_{K,M \geq 1} \mathcal{F}_{K,M} \quad ;$$

let $\mathcal{A}^{\overline{B}} \subset \mathcal{A}$ be the sigma algebra generated by the process $\overline{B}$.

The above equality (8) proves that $\mathcal{F}$ is a *base* for $\mathcal{A}^{\overline{B}}$: it is stable by finite intersection, and it generates the sigma algebra $\mathcal{A}^{\overline{B}}$.

Consider again $K, M \geq 1$ integers, and events $F_i, G_j \in \mathcal{E}$ for $i = 0, \dots K$, $j = 0, \dots M$, and the event

$$C = \{Y_0 \in F_0, \dots Y_K \in F_K, Z_0 \in G_0, \dots Z_M \in G_M\} \in \mathcal{A} \quad ;$$

let $A \in \mathcal{F}_{K,M}$ non empty; we want to show that

$$\mathbb{P}(C \mid A) = \mathbb{P}(C)$$

this will prove that $(\overline{Y}, \overline{Z})$ are independent of $\overline{B}$, by arbitriness of $(F_i), (G_j), K, M$ and since $\mathcal{F}$ is a base for $\mathcal{A}^{\overline{B}}$.

We fix $(u_i), (v_j)$ and define A as in equation (7); we let $N = \max\{u_K, v_M\}$ and define (b_n) as explained after equation (8). By defining

$$S^1 \stackrel{\text{def}}{=} S, S^0 \stackrel{\text{def}}{=} E \setminus S$$

for notation convenience, we can write equation (8) as

$$A = \{X_0 \in S^{b_0}, \dots X_N \in S^{b_N}\} \quad .$$

Let $E_0 \dots E_N \in \mathcal{E}$ be defined by

$$E_n \stackrel{\text{def}}{=} \begin{cases} F_k & \text{if } n = u_k \text{ for a } k \leq K \\ G_m & \text{if } n = v_m \text{ for a } m \leq M \\ E & \text{else} \end{cases}$$

then we compute

$$\begin{aligned}
\mathbb{P}(C \mid A) &= \mathbb{P}(\{X_{u_0} \in F_0, \dots X_{u_K} \in F_K, X_{v_0} \in G_0, \dots X_{v_M} \in G_M\} \mid \{X_0 \in S^{b_0}, \dots X_N \in S^{b_N}\}) = \\
&= \frac{\mathbb{P}\{X_{u_0} \in F_0, \dots X_{u_K} \in F_K, X_{v_0} \in G_0, \dots X_{v_M} \in G_M \ , \ X_0 \in S^{b_0}, \dots X_N \in S^{b_N}\}}{\mathbb{P}\{X_0 \in S^{b_0}, \dots X_N \in S^{b_N}\}} = \\
&= \frac{\prod_{n=0}^N \mathbb{P}\{X_n \in S^{b_n} \cap E_n\}}{\prod_{n=0}^N \mathbb{P}\{X_n \in S^{b_n}\}} = \prod_{n=0}^N \mathbb{P}(X_n \in E_n | X_n \in S^{b_n}) = \\
&= \prod_{k=0}^K \mu(F_k \mid S^1) \prod_{m=0}^M \mu(G_m \mid S^0) \quad ;
\end{aligned}$$

the last equality is due to the fact that: when $n = u_k$ then $b_n = 1$, when $n = v_m$ then $b_n = 0$, and for all other n we have $E_n = E$. Since the last term does not depend on A, that is, on $(u_i), (v_j)$, we obtain that $(\overline{Y}, \overline{Z})$ are independent of $\overline{B}$.

The above equality then also shows that

$$\mathbb{P}\{Y_0 \in F_0, \dots Y_K \in F_K, Z_0 \in G_0, \dots Z_M \in G_M\} = \prod_{k=0}^{K} \mu(F_k \mid S) \prod_{m=0}^{M} \mu(G_m \mid S^c)$$

and this implies that $\overline{Y}, \overline{Z}$ are processes of independent variables, distributed as in the thesis. By associativity of the independence, we conclude that the random variables B_n, Y_n, Z_n are independent. $\square$

3 Recycling in uniform random number generation

We now restrict our attention to the generation of uniformly distributed integer valued random variables. We will say that R *is a random variable of modulus* M when R is uniformly distributed in the range $0, \dots (M-1)$.

We present an algorithm, that we had thought of, and then found (different implementation, almost identical idea) in [1]. We present the latter implementation.

The following algorithm `Uniform random by bit recycling` in figure 1, given n, will return a random variable of modulus n; note that n can change between different calls to the algorithm.

```
 1: initialize the static integer variables $m = 1$ and $r = 0$
 2: procedure UNIFORM RANDOM BY BIT RECYCLING(n)
 3:     repeat
 4:         while $m < N$ do                          ▷ fill in the state
 5:             r : = 2*r + NextBit();
 6:             m : = 2*m;                            ▷ r is a random variable of modulus m
 7:         end while
 8:         q := $\lfloor m/n \rfloor$;               ▷ integer division, rounded down
 9:         if $r < n * q$ then
10:             d : = $r \bmod n$                     ▷ remainder, is a random variable of modulus n
11:             r : = $\lfloor r/n \rfloor$           ▷ quotient, is random variable of modulus q
12:             m : = q
13:             return d
14:         else
15:             r : = r - n*q                         ▷ r is still a random variable of modulus m
16:             m : = m - n*q                         ▷ the procedure loops back to line 3
17:         end if
18:     until forever
19: end procedure
```

Figure 1: Algorithm `Uniform random by bit recycling`

It uses two internal integer variables, m and r, which are not reset at the beginning of the algorithm (in C, you would declare them as "static"). Initially, $m = 1$ and $r = 0$.

The algorithm has an internal constant parameter N, which is a large integer such that $2N$ can still be represented exactly in the computer. We must have $n < N$. [3] The algorithm draws randomness from a function `NextBit()` that returns a random bit.

Here is an informal discussion of the algorithm, in the words of the original author [1]. *At line 10, as r is between 0 and* $(n * q - 1)$*, we can consider r as a random variable of modulus* $n * q$*. As this is divisible by n, then* $d := (r \bmod n)$ *will be uniformly distributed, and the quotient* $\lfloor r/n \rfloor$ *will be uniformly distributed between 0 and* $q - 1$*.*

Note that the theoretical running time is unbounded; we will though show in the next section that an accurate choice of parameters practically cancels this problem.

[3]We will show in next section that it is best to have $n << N$.

4 Mathematical analysis of the efficiency

We recall this simple idea.

Lemma 5 *Suppose R is a random variable of modulus MN; we perform the integer division $R = QN+D$ where $Q \in \{0, \dots (M-1)\}$ is the quotient and $D \in \{0, \dots (N-1)\}$ is the remainder; then Q is a random variable of modulus M and D is a random variable of modulus N; and Q, D are independent.*

Proposition 6 *Let us assume that repeated calls of `NextBit()` return a sequence of independent equidistributed bits. Then the above algorithm `Uniform random by bit recycling` in figure 1 on the previous page will return a sequence of independent and uniformly distributed numbers.*

Proof. We sketch the proof. We use the lemma above 5 and the theorem 4. Consider the notations in the second section. The bits returned by the call `NextBit()` builds up the process $\overline{X}$. When reaching the `if` (line 9 in the pseudocode at page 5), the choice $r < n * q$ is the choice of the value of B_n in equation (1). This (virtually) builds the process $\overline{B}$. At line 10 r is a variable in the process $\overline{Y}$; since it is of modulus nq, we return (using the lemma) the remainder as d, that is a random variable of modulus n, and push back the quotient into the state. At line 16 we would be defining a variable in the process $\overline{Z}$, that we push back into the state. □

The "pushing back" of most of the entropy back into the state recycles the bits, and improves greatly the efficiency.

The only wasted bits are related to the fact that the algorithm is throwing away the mathematical stream $\overline{B}$. Theoretically, if this stream would be feeded back into the state (for example, by employing Shannon-Fano-Elias coding), then efficiency would be exactly 100%.[4] Practically, the numbers N and n can be designed so that this is totally unneeded.

Remark 7 *Indeed, consider the implementation (see the code in the next sections) where the internal state is stored as 64bit unsigned integers, whereas n is restricted to be 32bit unsigned integer; so the internal constant is $N = 2^{62}$ while $n \in \{2 \dots 2^{32}-1\}$, When reaching the `if` at line 9, m is in the range $2^{62} \leq m < 2^{64}$, and r is uniform of modulus m; but $m - n * \lfloor m/n \rfloor$ is less than n, that is, less than 2^{32}; so the probability that $r \geq n * q$ at the `if` is less than $1/2^{30}$.*

In particular, this means that each B_n in the mathematical stream $\overline{B}$ contains $\sim 10^{-8}$bits of entropy, so there is no need to recycle them.

Indeed, in the numerical experiments we found out that the following algorithm wastes ~ 30 input bits on a total of $\sim 10^9$ input bits (!) this is comparable to the entropy of the internal state (and may also be due to numerical error in adding up $\log_2()$ values).

Also, this choice of parameters ensures that the algorithm will never practically loop twice before returning. When the condition in the `if` at line 9 is false, we will count it as a **failure**. In $\sim 10^{10}$ calls to the algorithm, we only experienced 3 failures. [5]

5 Speed, simple *vs* complex algorithms

We now consider the algorithm `Uniform random simple` in 2 on the following page.

Again, when the condition in the `if` at line 5 is false, we will count it as a **failure**.

This algorithm will always call the original RNG to obtain b bits, regardless of the value of n. When the algorithm fails, it starts again and again draws b bits. This is inefficient in terms of entropy: for small values of n it will produce far less entropy than it consume. But, will it be slower or faster than our previous algorithm? It turns out that the answer pretty much depends on the speed of the back-end RNG (and this is unsurprising); but also on how much time it takes to compute the basic operations "integer multiplications" $q * n$ and "integer division" $\lfloor N/n \rfloor$: we will see that, in some cases, these operations are so slow that they defeat the efficiency of the algorithm `Uniform random by bit recycling`.

[4]But this would render difficult to prove that the output numbers are independent...

[5]For this reason, the `else` block may be omitted with no big impact on the quality of the output – we implement this idea in the algorithm `uniform_random_by_bit_recycling_cheating`.

```
procedure UNIFORM RANDOM SIMPLE(n)
    repeat
        r : = GetRandomBits(b);                          ▷ fill the state with b bits
        q = $\lfloor N/n \rfloor$;                        ▷ integer division, rounded down
        if $r < n * q$ then
            return $r \bmod n$          ▷ remainder, is random variable of modulus n
        end if                                    ▷ otherwise, start all over again
    until forever
end procedure
```

Figure 2: `Uniform random simple` ; in our tests $N = 2^b$ or $N = 2^b - 1$, whereas $b = 32, 40, 48, 64$

6 Numerical tests

6.1 Architectures

The tests were performed in six different architectures,

(HW1) *Intel® Core ™ 2 Duo CPU E7500 2.93GHz*, in i686 mode,

(HW2) *Intel® Core ™ 2 Duo CPU P7350 2.00GHz*, in i686 mode,

(HW3) *AMD Athlon™ 64 X2 Dual Core Processor 4200+*, in x86_64 mode,

(HW4) *AMD Athlon ™ 64 X2 Dual Core Processor 4800+*, in x86_64 mode,

(HW5) *Intel® Core ™ 2 Duo CPU P7350 2.00GHz*, in x86_64 mode,

(HW6) *Intel ® Xeon ® CPU 5160 3.00GHz* , in x86_64 mode.

In the first five cases, the host was running a *Debian GNU/Linux* or *Ubuntu* O.S. , and the code was compiled using *gcc 4.4*, with the optimization flags
`-march=native -O3 -finline-functions -fno-strict-aliasing -fomit-frame-pointer -DNDEBUG .`
In the last case, the O.S. was *Gentoo* and the code was compiled using *gcc 4.0* with flags
`-march=nocona -O3 -finline-functions -fno-strict-aliasing -fomit-frame-pointer -DNDEBUG .`

6.2 Back-end PRNGs

To test the speed of the following algorithms, we used four different back-end PRNGs.

(sfmt_sse) The *SIMD oriented Fast Mersenne Twister(SFMT)* ver. 1.3.3 by Mutsuo Saito and Makoto Matsumoto [2] (compiled with SSE support)

(xorshift) The *xorshift* generator by G. Marsaglia [4]

(sfmt_sse_md5) as sfmt_sse above, but moreover the output is cryptographically protected using the `MD5` algorithm

(bbs260) The Blum-Blum-Shub algorithm [5], with two primes of size $\sim$ 130bit.

The last two were home-made, as examples of slower but (possibly) cryptographically strong RNG [6]. All of the above were uniformized to implement two functions, `my_gen_rand32()` and `my_gen_rand64()`, that return (respectively) a 32bit or a 64bit unsigned integer, uniformly distributed. The C code for all the above is in the appendix B.1. The speeds of the different RNGs are listed in the tables in sec. A.1 on page 10.

We also prepared a simple *counter* "RNG" algorithm, that returns numbers that are in arithmetic progression; since it is very simple, it is useful to assess the overhead complexity in the testing code itself; this overhead is on the order of 2 to to 4 *ns*, depending on the CPUs.

[6]The author makes no guarantees, though, that the implemented versions are really good and cryptographically strong RNGs — we are interested only in their speeds.

6.3 *Ad hoc* functions

We implemented some *ad hoc* functions, that are then used by the uniform RNGs (that are described in the next section).

NextBit returns a bit

Next2Bit returns two bits

NextByte returns 8 bits

NextWord returns 16 bits

For any of the above, we prepared many variants, that internally call either the `my_gen_rand32()` or `my_gen_rand64()` calls (see the C code in sec. B.2 on page 20) and then we benchmarked them in all architecture, to choose the faster one (that is then used by the uniform RNGs). [7]

We also prepared a specific method (that is not used for the uniform RNGs):

NextCard returns a number uniformly distributed in the range $0 \dots 51$ (it may be thought of as a card randomly drawn from a deck of cards).

The detailed timings are in the tables in sec. A.1 on page 10.

6.4 Uniform RNGs

We implemented nine different versions of uniform random generators. The C code is in B.3 on page 23; we here briefly describe the ideas. Four versions are based on the "simple" generator in fig. 2:

uniform_random_simple32 uses 32bit variables internally, $N = 2^{32} - 1$, and consumes a 32bit random number, (a call to `my_gen_rand32()`) each time

uniform_random_simple40 uses 64bit variables internally, $N = 2^{40}$, and calls `my_gen_rand32()` and `NextByte()` each time

uniform_random_simple48 uses 64bit variables internally, $N = 2^{48}$, and calls `my_gen_rand32()` and `NextWord()` each time

uniform_random_simple64 uses 64bit variables internally, $N = 2^{64} - 1$, and calls `my_gen_rand64()` each time.

Then there are three versions based on the "bit recycling" generator in fig. 1 (all use 64bit variables internally):

uniform_random_by_bit_recycling is the code in fig. 1 (but it refills the state by popping two bits at a time)

uniform_random_by_bit_recycling_faster it refills the state by popping words, bytes and pairs of bits, for improved efficiency

uniform_random_by_bit_recycling_cheating as the "faster" one, but the *if/else* block is not implemented, and the modulus $r \bmod n$ is always returned; this is not mathematically exact, but the probability that it is inexact is $\sim 2^{-30}$.

Moreover there are "mixed" methods

uniform_random_simple_recycler uses 32bit variables internally, keeps an internal state that is sometimes initialized but not refilled each time (so, it is useful only for small n),

uniform_random_by_bit_recycling_32 when $n < 2^{29}$, it implements the "bit recycling" code using 32bit variables; when $2^{29} \leq n < 2^{32}$, it implements a "simple"–like method, using only bit shifting.

We tested them in all of the architectures, for different values of the modulus n, and graphed the results (see appendix A.2.1 on page 11).

[7] We had to make an exception for when the back-end RNG is based on `SFMT`, since `SFMT` cannot mix 64bit and 32bit random number generations: in that case, we forcibly used the 32bit versions (that in most of our benchmarks are anyway slightly faster).

6.5 Timing

To benchmark the algorithms, we computed the process time using both the Posix call `clock()` (that returns an approximation of processor time used by the program) and the CPU `TSC` (that counts the number of CPU ticks). When benchmarking one of the above back-end RNGs or the *ad hoc* functions, we called it in repeated loops of 2^{24} iterations each, repeating them for at least 1 second of processor time; and then compared the data provided by `TSC` and `clock()`. We also prepared a statistics of the values

$$\texttt{cycles\_per\_clock} := \frac{\Delta\texttt{TSC}}{\Delta\texttt{clock()}}$$

so that we could convert CPU cycles to nanoseconds; we verified that the standard deviation of the logarithm of the above quantity was usually less than 1%.

To avoid over-optimization of the compiler, the results of any benchmarked function was *xor*-ed in a *bucket* variable, that was then printed on screen.

During benchmarking, we disabled the CPU power-saving features, forcing the CPU to be at maximum performance (using the `cpufreq-set` command) and also we tied the process to one core (using the `taskset` command).

Unfortunately the `clock()` call, in GNU/Linux systems, has a time resolution of $0.01sec$, so it was too coarse to be used for the graphs in section A.2.1: for those graphs, only the `TSC` was used (and then cycles were converted to nanoseconds, using the average value of `cycles_per_clock`).

6.6 Conclusions

While efficiency is exactly mathematically assessed, computational speed is a more complex topic, and sometimes quite surprising. We report some considerations.

1. In our Intel^{TM} CPUs running in 32bit mode, integers divisions and remainder computation using 64bit variables are quite slow: in `HW1`, each such operations cost $\sim 15ns$.

2. Any bit recycling method that we could think of needs at least four arithmetic operations for each result it produces; moreover there is some code to refill the internal state.

3. In the same CPUs, the cost of *bit shifting* or *xor* operations are on the order of 3 ns, even on 64bit variables; moreover the back-ends `sfmt_sse` and `xorshift` can produce a 32bit random number in $\sim$ 5 ns.

4. So, unsurprisingly, when the back end is `sfmt_sse` and `xorshift`, and the Intel^{TM} CPU runs in 32bit mode, the fastest methods are the “simple32” and “simple_recycler” methods, that run in $\sim 10ns$; and the bit recycling methods are at least 5 times slower than those.

5. When the back-end is `sfmt_sse` and `xorshift`, but the the Intel^{TM} CPU runs in 64bit mode, the fastest method are still the “simple32” and “simple_recycler” methods; the bit recycling methods are twice slower.

6. When the back-ends RNGs are `sfmt_sse` or `xorshift`, in the AMD^{TM} CPUs, the “simple32” and “simple_recycler” take $\sim 40ns$; this is related to the fact that 32bit division and remainder computation need $\sim 20ns$ (as is shown in sec. A.2.2). So these methods are much slower than the back-ends RNGs, that return a 32bit random number in $\sim 6ns$.

   One consequence is that, since the `uniform_random_by_bit_recycling_32` for $n > 2^{29}$ uses a “simple”–like method with only bit shifting, then it is much faster than the “simple32” and “simple_recycler”.

   What we cannot explain is that, in the same architectures, the `NextCard32` function, that implements the same type of operations, runs in $\sim 8ns$ (!)

   (We also tried to test the above with different optimizations. Using the `xorshift` back-end, setting optimization flags to be just `-O0`, `NextCard32` function takes $\sim 21ns$; setting it to `-O`, it takes $\sim 12ns$.)

7. The back-end RNGs `sfmt_sse_md5` or `bbs260`, are instead much slower, that is, `sfmt_sse_md5` needs ~ 300 *ns* to produce a 32bit number, and `bbs260` needs ~ 500 *ns* when the CPU runs in 64bit mode and more than a microsecond (!) in 32bit mode.

   In this case, the bit recycling methods are usually faster. Their speed is dominated by how many times the back-end RNGs is called, so it can be estimated in terms of *entropy bitrate*, and indeed the graphs are (almost) linear (since the abscissa is in logarithmic scale).

8. One of the biggest surprises comes from the `NextCard` functions: there are four implementations,

   - using 64bit or 32bit variables;
   - computing a result for each call, or precomputing them and storing in an array (the "prefilled" versions).

   The speed benchmarks give discordant results. When using the faster back-ends `sfmt_sse` and `xorshift`, the "prefilled" versions are slower. When using the slower back-ends `sfmt_sse_md5` or `bbs260`, the 64bit "prefilled" version is the fastest in Intel^{TM} CPUs; but it is instead much slower than the "non prefilled" version in AMD^{TM} CPUs. It is possible that the cache misses are playing a rôle in this, but we cannot provide a good explanation.

9. Curiously, in some Intel^{TM} CPUs, the time needed for an integer arithmetic operation depends also on the *values* of the operands (and not only on the bit sizes of the variable)! See the graph in sec. A.2.2. So, the speed of the functions depend on the value of the modulus n. This is the reason why some the graphs are all oscillating in nature.

   In particular, when we looked at the graphs for *Core 2* architectures in 32bit mode, by looking at the graphs of the functions `simple_40`, `simple_48` (where $N = 2^{40}, 2^{48}$ constant) we noted that the operations $q := N/n, qn := n * q$ are ~ 10 *ns* slower when $n < N2^{-32}$ than when $n > N2^{-32}$. This is similar to what is seen in the graphs in sec. A.2.2.

   Instead the speed graphs in AMD^{TM} CPUs are almost linear, and this is well explained by the average number of needed operations.

Summarizing, the speeds are quite difficult to predict; if a uniform random generator is to be used for n in a certain range, and the back-end RNG takes approximatively as much time as 4 integer operations in 64bits, then the only sure way to decide which algorithm is the fastest one is by benchmarking. If a a uniform random generator is to be used for a constant and specific n (such as in the case of the `NextCard` function), there may be different strategies to implement it, and again the only sure way to decide which algorithm is the fastest one is by benchmarking.

A Test results

A.1 Speed of back-end RNGs and *Ad hoc* functions

These tables list the average time (in nanoseconds) of the back-end RNGs and the *ad hoc* functions (see the C code in sec. B.2 on page 20); these same data are plotted as red crosses in the plots of the next section. For each family, the fastest function is marked blue; competitors that differ less than 10% are italic and blue; competitors that are slower more than 50% are red.

	sfmt_sse						xorshift					
	HW1	HW2	HW3	HW4	HW5	HW6	HW1	HW2	HW3	HW4	HW5	HW6
Next2Bit32	2.6	3.9	4.9	*4.8*	*3.9*	*2.6*	2.5	3.7	*4.7*	*4.2*	3.7	2.8
Next2Bit64	4.2	4.7	*5.0*	4.4	3.7	2.5	3.3	6.3	4.5	4.0	3.7	2.4
NextBit32	2.5	3.7	4.6	4.2	*3.7*	*2.8*	2.5	3.6	*4.4*	*3.9*	3.6	2.4
NextBit32_by_mask	2.5	3.7	*4.9*	*4.4*	*3.7*	*2.8*	2.5	3.6	*4.4*	*3.9*	3.6	2.7
NextBit64	3.2	4.6	*4.7*	4.2	3.6	2.8	3.2	4.7	4.3	3.8	3.6	2.4
NextByte32	3.5	5.1	6.1	5.4	5.1	3.8	*3.0*	*4.4*	5.1	4.5	*4.5*	3.3
NextByte64	*3.3*	5.6	*6.5*	*5.8*	4.4	3.1	*3.0*	*4.4*	6.2	5.2	4.3	2.8
NextByte64_prefilled	3.1	4.6	*6.3*	*5.6*	*4.5*	3.4	2.9	4.3	5.8	6.0	*4.6*	*3.0*
NextCard32	3.5	5.1	8.0	7.1	5.5	7.0	3.4	4.9	5.6	5.0	5.2	6.9
NextCard32_prefilled	6.1	9.0	24.4	21.7	11.4	7.1	6.0	15.6	22.2	19.7	11.1	7.1
NextCard64	22.0	32.2	6.5	5.8	7.0	4.9	22.4	32.8	6.6	5.8	7.0	4.9
NextCard64_prefilled	22.0	32.2	37.9	33.5	20.5	*5.4*	24.0	37.0	37.4	33.1	20.4	*5.3*
NextWord32	4.2	6.2	7.5	6.7	6.5	4.5	3.4	5.0	5.7	5.0	*5.0*	3.3
NextWord64	4.2	*6.5*	6.5	5.8	5.2	3.7	4.2	5.9	5.7	5.0	4.9	*3.4*
my_gen_rand32	3.7	5.4	6.5	5.7	5.4	3.9	3.5	5.1	5.0	4.4	5.4	3.5
my_gen_rand64	4.2	6.3	8.4	7.4	6.3	5.2	4.5	6.9	6.8	6.0	7.4	4.8

	sfmt_sse_md5						bbs260					
	HW1	HW2	HW3	HW4	HW5	HW6	HW1	HW2	HW3	HW4	HW5	HW6
Next2Bit32	18.7	27.5	30.2	26.7	32.0	23.1	55.2	81.0	32.9	31.4	36.3	28.6
Next2Bit64	11.6	17.0	17.6	15.6	18.1	12.9	35.6	52.2	18.9	17.5	20.0	15.2
NextBit32	10.5	15.5	17.2	15.2	17.8	12.6	28.7	*42.2*	19.4	16.4	30.4	15.3
NextBit32_by_mask	10.3	15.2	17.6	15.6	18.2	12.9	28.7	*42.4*	18.9	16.8	25.0	15.5
NextBit64	7.5	10.9	11.1	9.8	11.1	7.8	19.7	41.5	11.7	10.4	11.8	9.8
NextByte32	67.6	99.5	108.1	95.7	117.9	84.8	212.5	315.5	118.2	113.7	136.3	102.7
NextByte64	*36.2*	*53.5*	57.1	50.5	60.5	43.8	130.6	192.9	62.4	*59.7*	69.2	*56.2*
NextByte64_prefilled	35.7	52.5	56.9	50.3	60.6	43.7	131.0	192.7	62.6	56.4	69.7	54.1
NextCard32	56.2	83.3	88.1	77.9	96.5	71.7	173.9	255.6	97.9	85.9	110.4	86.4
NextCard32_prefilled	60.7	89.7	104.3	92.2	102.9	*37.6*	176.3	259.6	115.6	106.3	117.5	45.3
NextCard64	46.4	68.1	43.5	38.4	49.1	34.8	119.0	193.4	48.5	43.1	55.3	40.7
NextCard64_prefilled	46.0	67.5	74.7	66.0	62.0	*35.2*	118.7	173.6	79.2	72.9	68.1	*43.8*
NextWord32	132.7	195.7	209.2	185.4	228.3	166.7	429.4	633.0	236.0	204.2	265.3	202.9
NextWord64	69.3	102.8	108.1	95.6	117.8	84.5	264.2	387.4	117.9	105.6	136.2	108.7
my_gen_rand32	262.5	391.5	413.3	366.7	457.3	328.7	859.8	1263.3	*459.2*	403.1	525.4	399.5
my_gen_rand64	263.0	390.4	413.1	368.4	457.7	328.4	1018.1	1513.1	454.3	402.9	523.9	398.5

	counter					
	HW1	HW2	HW3	HW4	HW5	HW6
Next2Bit32	2.5	3.6	*4.6*	*4.1*	*3.6*	*2.4*
Next2Bit64	3.4	4.9	4.4	3.9	3.6	2.4
NextBit32	2.4	3.6	*4.3*	*3.8*	3.6	2.4
NextBit32_by_mask	2.4	3.6	*4.4*	*3.9*	3.6	2.4
NextBit64	3.1	4.6	4.2	3.7	3.6	2.7
NextByte32	*2.9*	4.3	4.8	4.2	4.3	2.8
NextByte64	*2.9*	4.3	5.3	4.7	*3.8*	*2.9*
NextByte64_prefilled	2.8	3.8	*5.2*	*4.6*	3.6	*3.0*
NextCard32	2.8	4.2	5.1	4.5	4.5	6.4
NextCard32_prefilled	5.2	7.7	21.9	19.4	8.9	6.9
NextCard64	22.0	32.7	6.0	5.4	6.0	4.2
NextCard64_prefilled	21.7	31.9	36.9	32.7	19.7	4.9
NextWord32	2.9	4.3	*5.0*	*4.4*	4.8	3.2
NextWord64	3.3	4.9	4.6	4.1	4.0	2.7
my_gen_rand32	2.0	3.0	3.6	3.2	3.5	2.0
my_gen_rand64	2.4	3.5	3.6	3.2	3.5	2.3

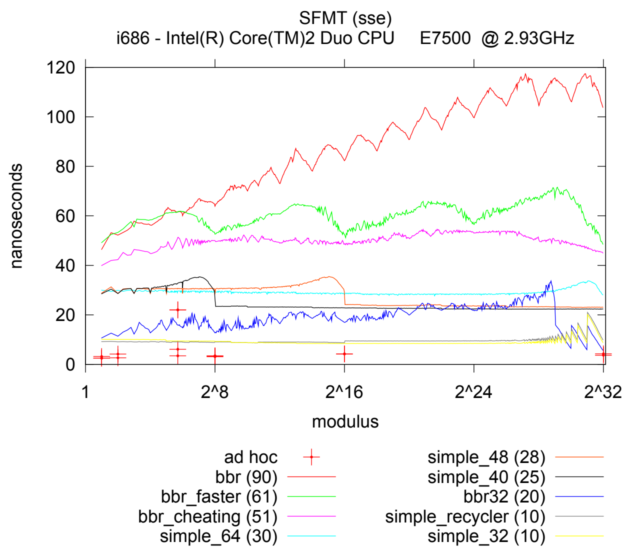
A.2 Graphs

In all of the following graphs, the abscissa is n, (that is the modulus of the uniform RNGs); the abscissa is in log-scale (precisely, it contains all n from 2 to 32, and then n is incremented by $\lfloor n/32 \rfloor$ up to 2^{32}, for a total of 733 samples). The number in parentheses near the graph labels are the average time for call (in nanoseconds; averaged in the aforementioned log scale).

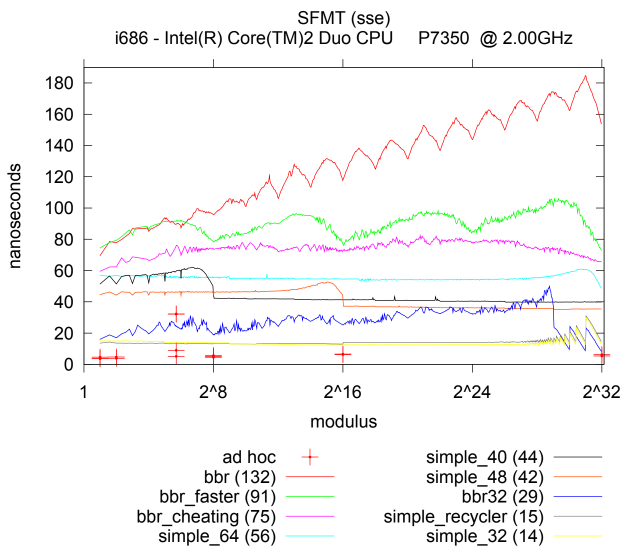
A.2.1 Uniform random generators

To reduce the size of the labels, we abbreviated `uniform_random_by_bit_recycling` as `bbr`, and `uniform_random_simple` as `simple`.

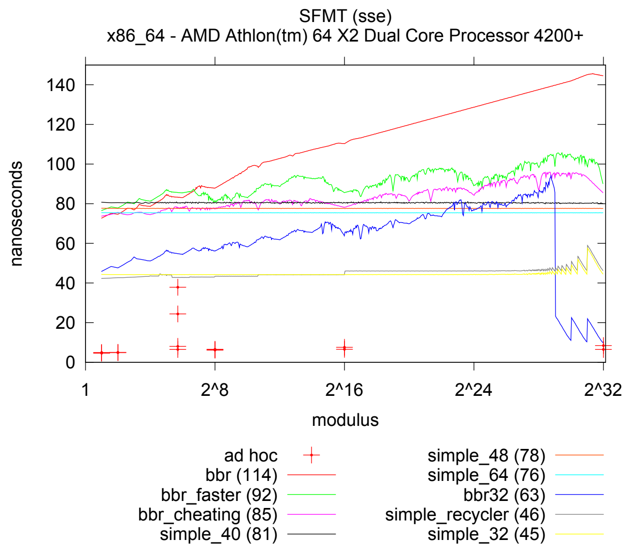
SFMT



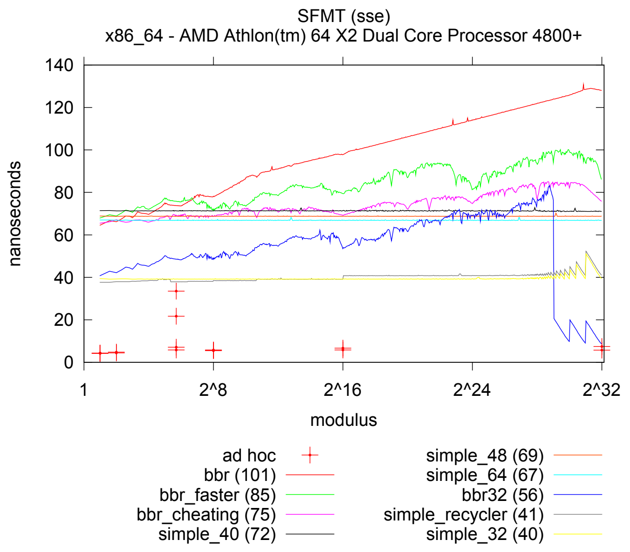
SFMT (sse)
i686 - Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz
nanoseconds
0
20
40
60
80
100
120
1
2^8
2^16
2^24
2^32
modulus
ad hoc
bbr (90)
bbr_faster (61)
bbr_cheating (51)
simple_64 (30)
simple_48 (28)
simple_40 (25)
bbr32 (20)
simple_recycler (10)
simple_32 (10)



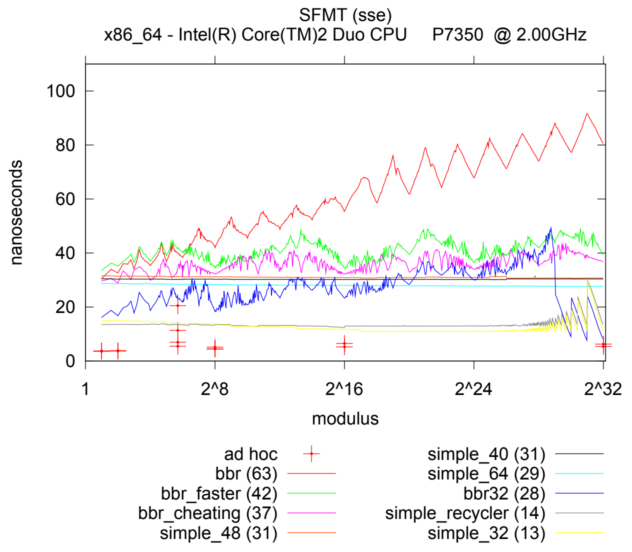
SFMT (sse)
i686 - Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
nanoseconds
0
20
40
60
80
100
120
140
160
180
1
2^8
2^16
2^24
2^32
modulus
ad hoc
bbr (132)
bbr_faster (91)
bbr_cheating (75)
simple_64 (56)
simple_40 (44)
simple_48 (42)
bbr32 (29)
simple_recycler (15)
simple_32 (14)



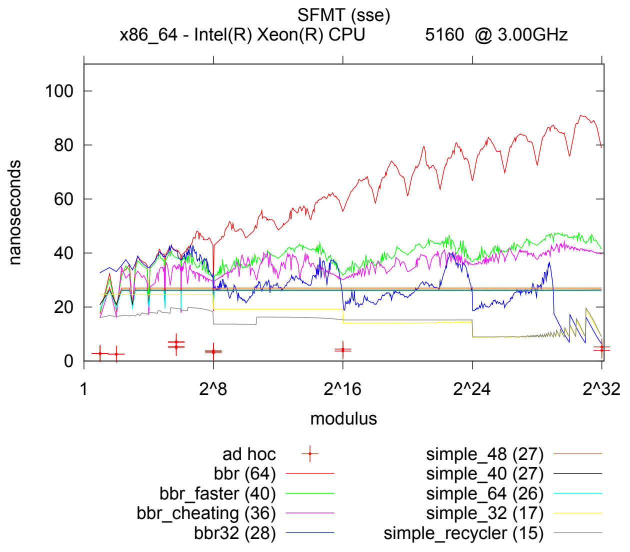
SFMT (sse)
x86_64 - AMD Athlon(tm) 64 X2 Dual Core Processor 4200+
nanoseconds
0
20
40
60
80
100
120
140
1
2^8
2^16
2^24
2^32
modulus
ad hoc
bbr (114)
bbr_faster (92)
bbr_cheating (85)
simple_40 (81)
simple_48 (78)
simple_64 (76)
bbr32 (63)
simple_recycler (46)
simple_32 (45)



SFMT (sse)
x86_64 - AMD Athlon(tm) 64 X2 Dual Core Processor 4800+
nanoseconds
0
20
40
60
80
100
120
140
1
2^8
2^16
2^24
2^32
modulus
ad hoc
bbr (101)
bbr_faster (85)
bbr_cheating (75)
simple_40 (72)
simple_48 (69)
simple_64 (67)
bbr32 (56)
simple_recycler (41)
simple_32 (40)

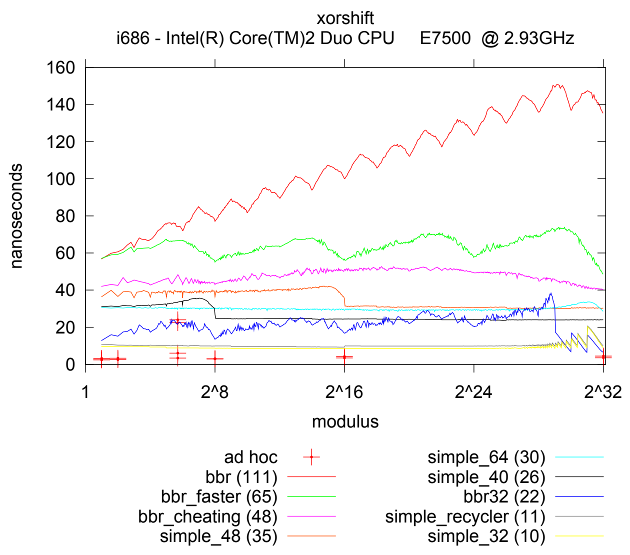


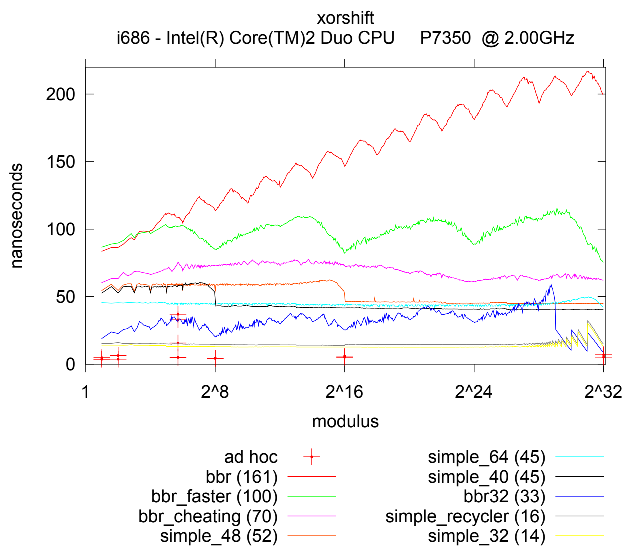
SFMT (sse)
x86_64 - Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
nanoseconds
0
20
40
60
80
100
1
2^8
2^16
2^24
2^32
modulus
ad hoc
bbr (63)
bbr_faster (42)
bbr_cheating (37)
simple_48 (31)
simple_40 (31)
simple_64 (29)
bbr32 (28)
simple_recycler (14)
simple_32 (13)

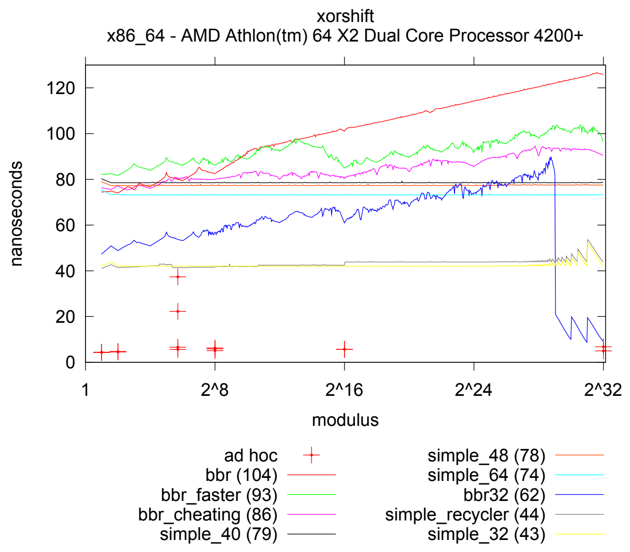


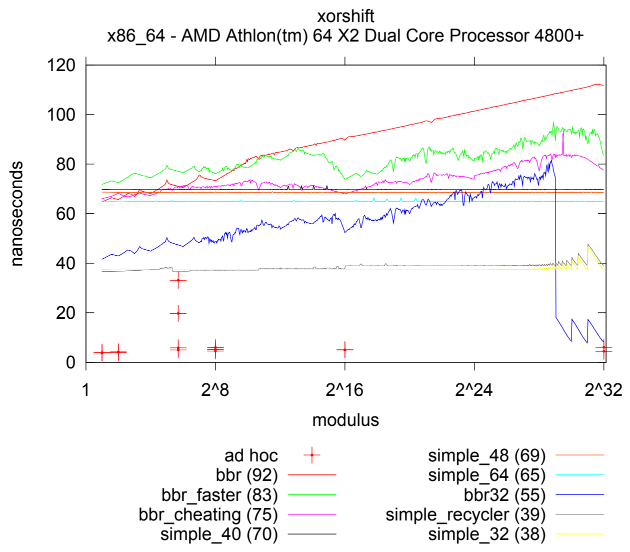
SFMT (sse)
x86_64 - Intel(R) Xeon(R) CPU 5160 @ 3.00GHz
nanoseconds
0
20
40
60
80
100
1
2^8
2^16
2^24
2^32
modulus
ad hoc
bbr (64)
bbr_faster (40)
bbr_cheating (36)
bbr32 (28)
simple_48 (27)
simple_40 (27)
simple_64 (26)
simple_32 (17)
simple_recycler (15)

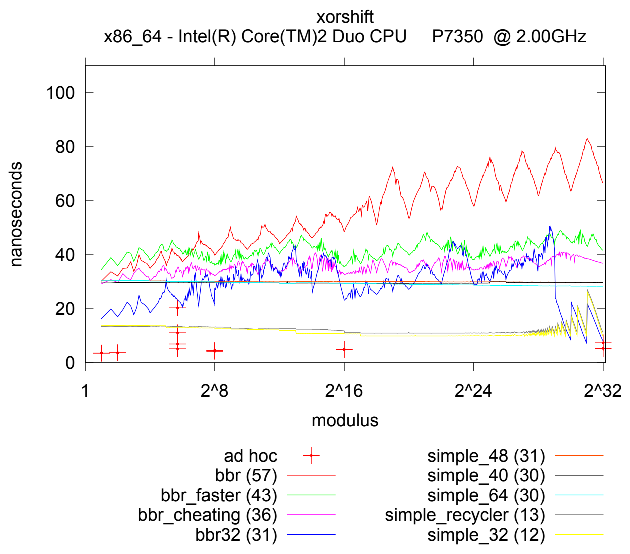
xorshift

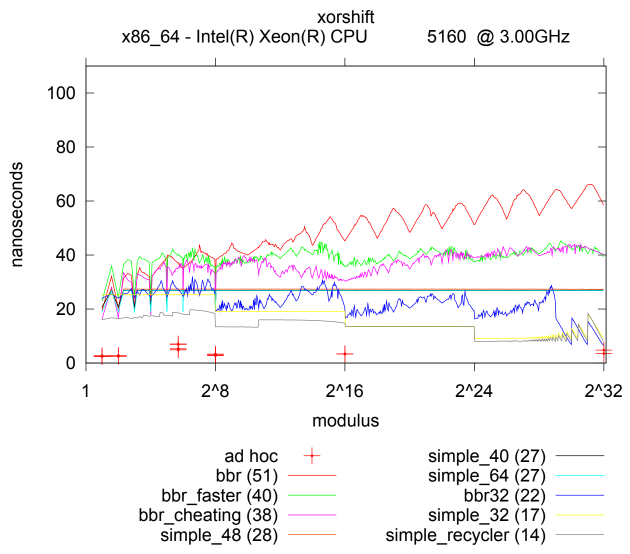

xorshift
i686 - Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
bbr (111)
bbr_faster (65)
bbr_cheating (48)
simple_48 (35)
simple_64 (30)
simple_40 (26)
bbr32 (22)
simple_recycler (11)
simple_32 (10)


xorshift
i686 - Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
bbr (161)
bbr_faster (100)
bbr_cheating (70)
simple_48 (52)
simple_64 (45)
simple_40 (45)
bbr32 (33)
simple_recycler (16)
simple_32 (14)

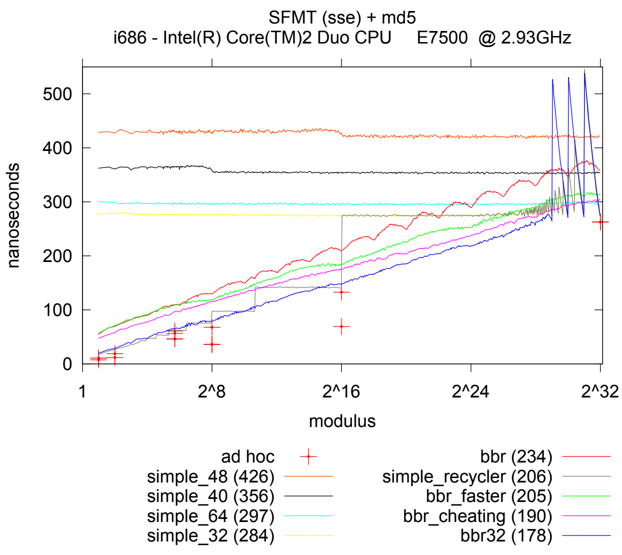

xorshift
x86_64 - AMD Athlon(tm) 64 X2 Dual Core Processor 4200+
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
bbr (104)
bbr_faster (93)
bbr_cheating (86)
simple_40 (79)
simple_48 (78)
simple_64 (74)
bbr32 (62)
simple_recycler (44)
simple_32 (43)

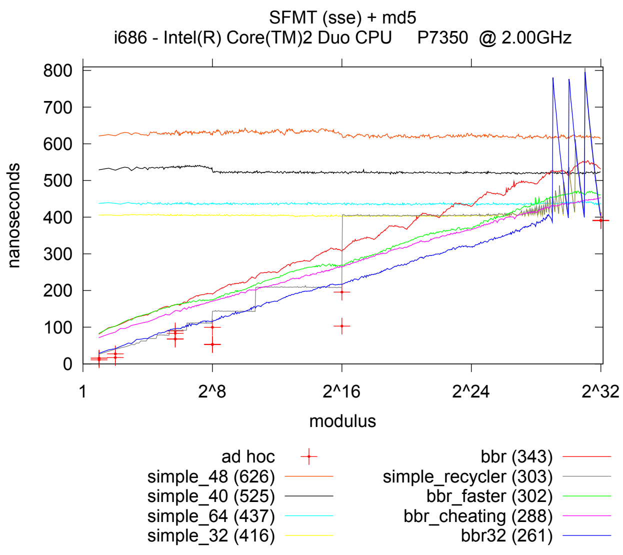

xorshift
x86_64 - AMD Athlon(tm) 64 X2 Dual Core Processor 4800+
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
bbr (92)
bbr_faster (83)
bbr_cheating (75)
simple_40 (70)
simple_48 (69)
simple_64 (65)
bbr32 (55)
simple_recycler (39)
simple_32 (38)

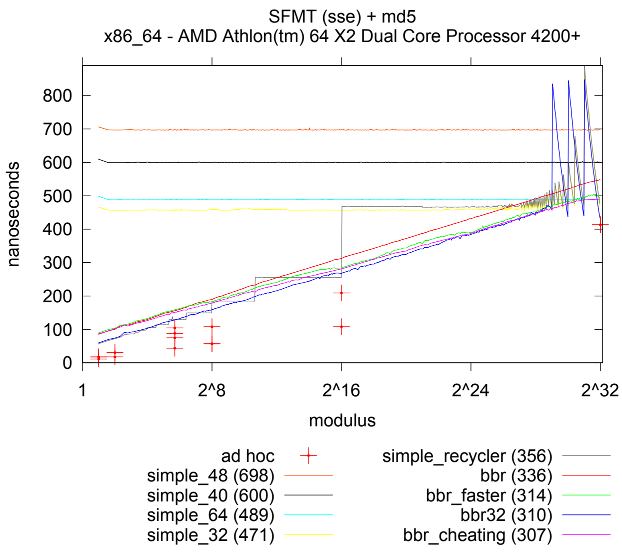

xorshift
x86_64 - Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
bbr (57)
bbr_faster (43)
bbr_cheating (36)
bbr32 (31)
simple_48 (31)
simple_40 (30)
simple_64 (30)
simple_recycler (13)
simple_32 (12)

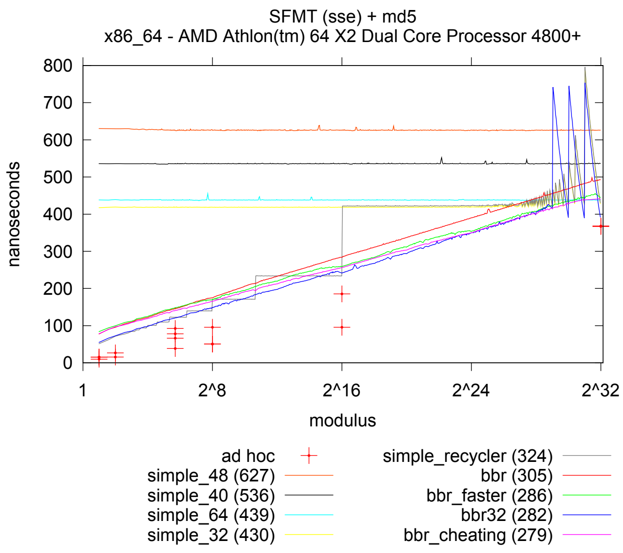

xorshift
x86_64 - Intel(R) Xeon(R) CPU 5160 @ 3.00GHz
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
bbr (51)
bbr_faster (40)
bbr_cheating (38)
simple_48 (28)
simple_40 (27)
simple_64 (27)
bbr32 (22)
simple_32 (17)
simple_recycler (14)

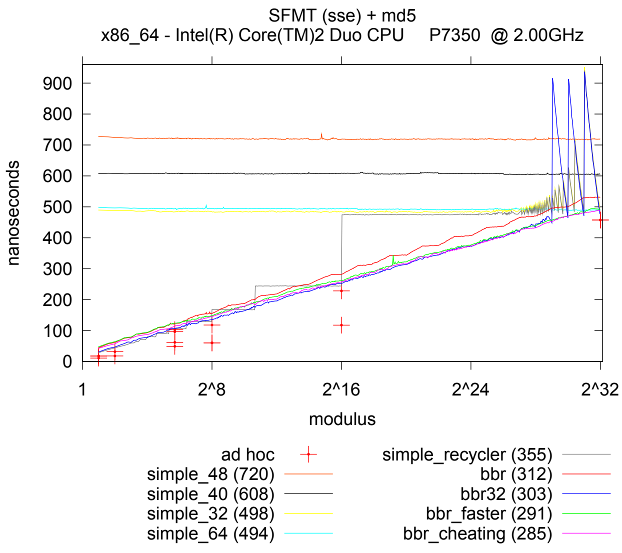
SFMT + MD5

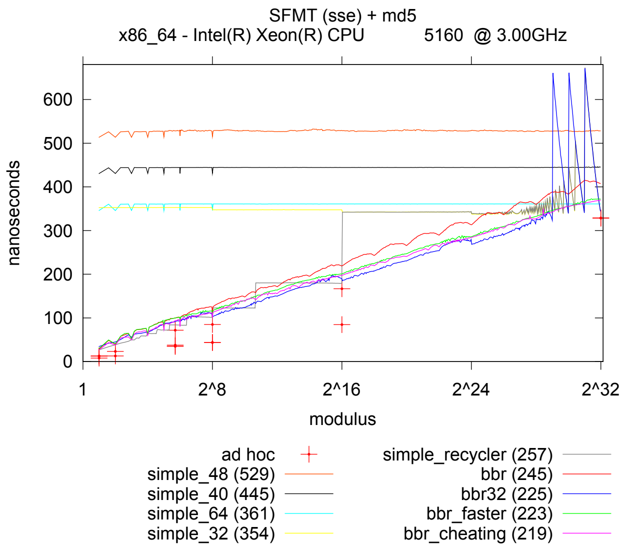

SFMT (sse) + md5
i686 - Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
simple_48 (426)
simple_40 (356)
simple_64 (297)
simple_32 (284)
bbr (234)
simple_recycler (206)
bbr_faster (205)
bbr_cheating (190)
bbr32 (178)


SFMT (sse) + md5
i686 - Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
simple_48 (626)
simple_40 (525)
simple_64 (437)
simple_32 (416)
bbr (343)
simple_recycler (303)
bbr_faster (302)
bbr_cheating (288)
bbr32 (261)

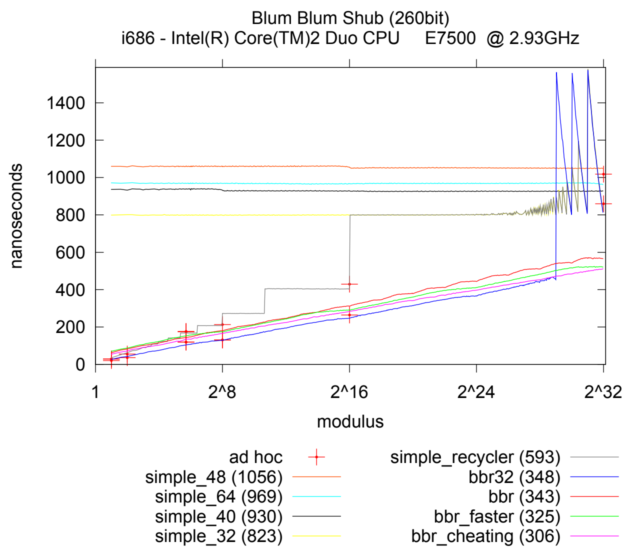

SFMT (sse) + md5
x86_64 - AMD Athlon(tm) 64 X2 Dual Core Processor 4200+
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
simple_48 (698)
simple_40 (600)
simple_64 (489)
simple_32 (471)
simple_recycler (356)
bbr (336)
bbr_faster (314)
bbr32 (310)
bbr_cheating (307)

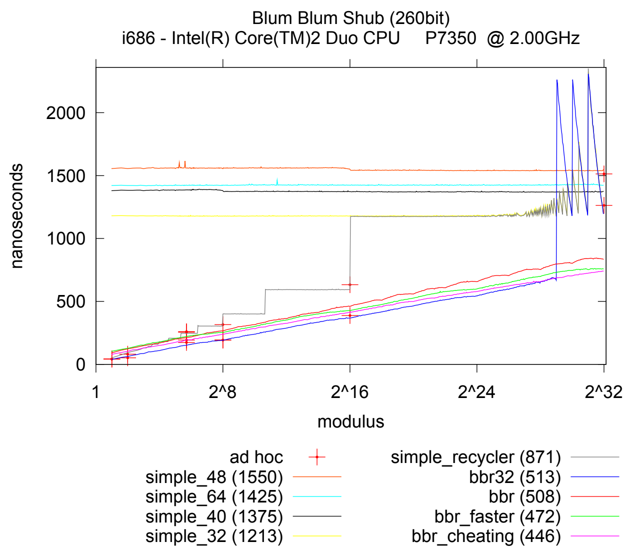

SFMT (sse) + md5
x86_64 - AMD Athlon(tm) 64 X2 Dual Core Processor 4800+
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
simple_48 (627)
simple_40 (536)
simple_64 (439)
simple_32 (430)
simple_recycler (324)
bbr (305)
bbr_faster (286)
bbr32 (282)
bbr_cheating (279)

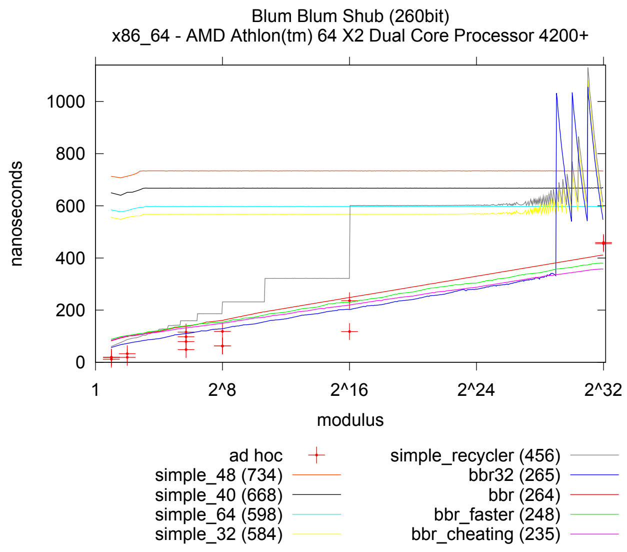

SFMT (sse) + md5
x86_64 - Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
simple_48 (720)
simple_40 (608)
simple_32 (498)
simple_64 (494)
simple_recycler (355)
bbr (312)
bbr32 (303)
bbr_faster (291)
bbr_cheating (285)

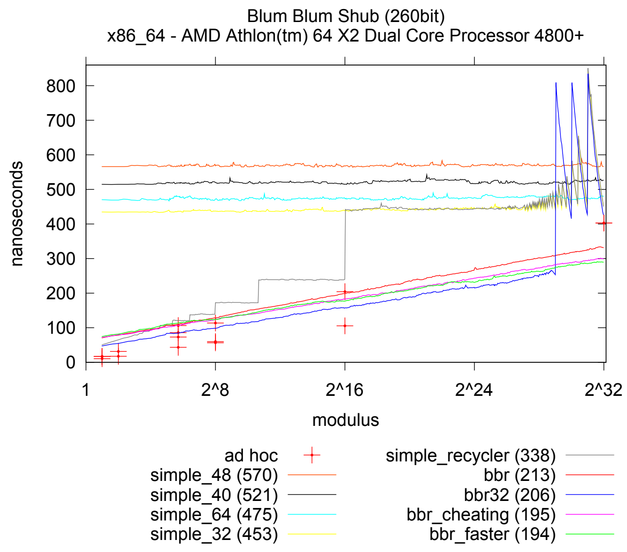

SFMT (sse) + md5
x86_64 - Intel(R) Xeon(R) CPU 5160 @ 3.00GHz
nanoseconds
modulus
1
2^8
2^16
2^24
2^32
ad hoc
simple_48 (529)
simple_40 (445)
simple_64 (361)
simple_32 (354)
simple_recycler (257)
bbr (245)
bbr32 (225)
bbr_faster (223)
bbr_cheating (219)

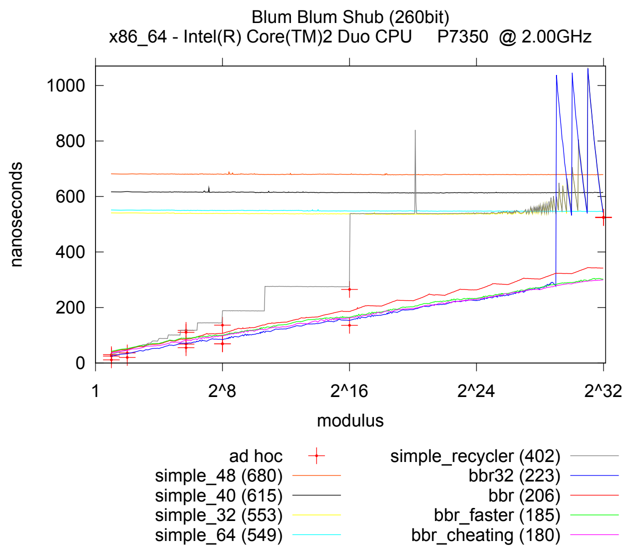
bbs260

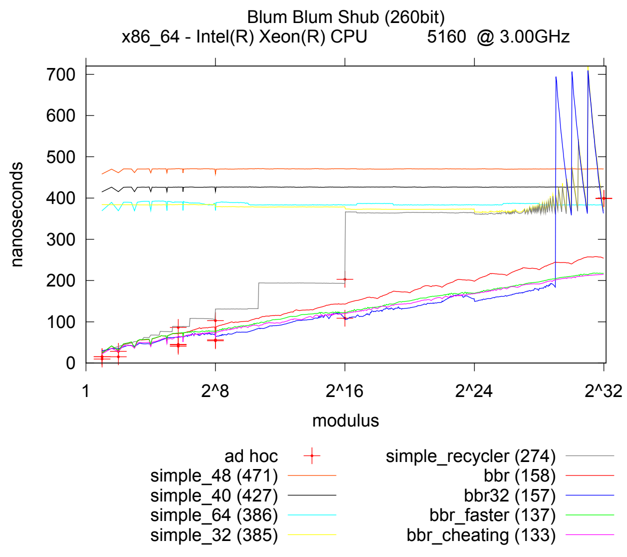

Blum Blum Shub (260bit)
i686 - Intel(R) Core(TM)2 Duo CPU E7500 @ 2.93GHz
nanoseconds
0
200
400
600
800
1000
1200
1400
1
2^8
2^16
2^24
2^32
modulus
ad hoc
simple_48 (1056)
simple_64 (969)
simple_40 (930)
simple_32 (823)
simple_recycler (593)
bbr32 (348)
bbr (343)
bbr_faster (325)
bbr_cheating (306)


Blum Blum Shub (260bit)
i686 - Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
nanoseconds
0
500
1000
1500
2000
1
2^8
2^16
2^24
2^32
modulus
ad hoc
simple_48 (1550)
simple_64 (1425)
simple_40 (1375)
simple_32 (1213)
simple_recycler (871)
bbr32 (513)
bbr (508)
bbr_faster (472)
bbr_cheating (446)


Blum Blum Shub (260bit)
x86_64 - AMD Athlon(tm) 64 X2 Dual Core Processor 4200+
nanoseconds
0
200
400
600
800
1000
1
2^8
2^16
2^24
2^32
modulus
ad hoc
simple_48 (734)
simple_40 (668)
simple_64 (598)
simple_32 (584)
simple_recycler (456)
bbr32 (265)
bbr (264)
bbr_faster (248)
bbr_cheating (235)

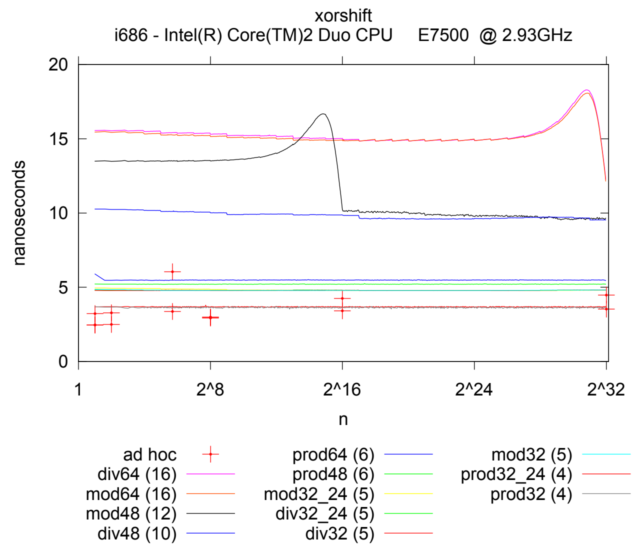

Blum Blum Shub (260bit)
x86_64 - AMD Athlon(tm) 64 X2 Dual Core Processor 4800+
nanoseconds
0
100
200
300
400
500
600
700
800
1
2^8
2^16
2^24
2^32
modulus
ad hoc
simple_48 (570)
simple_40 (521)
simple_64 (475)
simple_32 (453)
simple_recycler (338)
bbr (213)
bbr32 (206)
bbr_cheating (195)
bbr_faster (194)

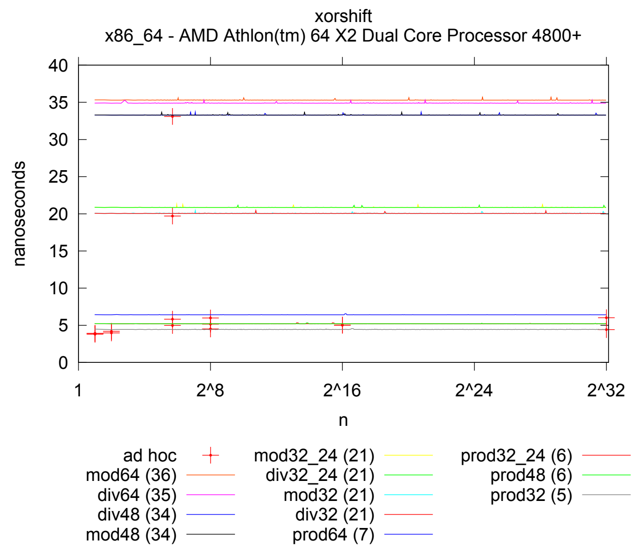

Blum Blum Shub (260bit)
x86_64 - Intel(R) Core(TM)2 Duo CPU P7350 @ 2.00GHz
nanoseconds
0
200
400
600
800
1000
1
2^8
2^16
2^24
2^32
modulus
ad hoc
simple_48 (680)
simple_40 (615)
simple_32 (553)
simple_64 (549)
simple_recycler (402)
bbr32 (223)
bbr (206)
bbr_faster (185)
bbr_cheating (180)

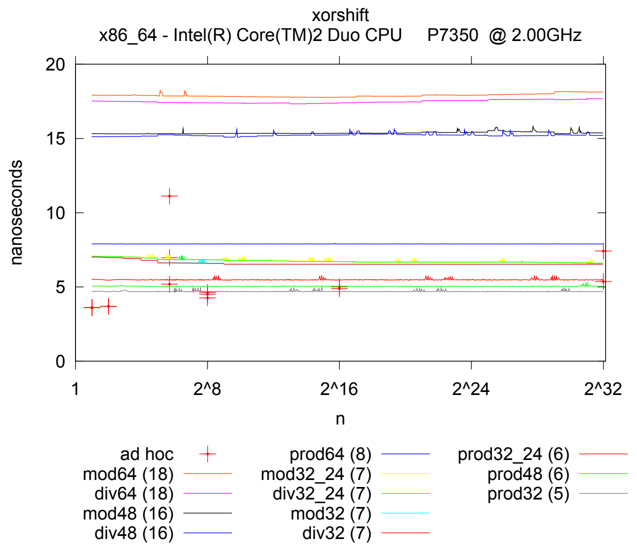

Blum Blum Shub (260bit)
x86_64 - Intel(R) Xeon(R) CPU 5160 @ 3.00GHz
nanoseconds
0
100
200
300
400
500
600
700
1
2^8
2^16
2^24
2^32
modulus
ad hoc
simple_48 (471)
simple_40 (427)
simple_64 (386)
simple_32 (385)
simple_recycler (274)
bbr (158)
bbr32 (157)
bbr_faster (137)
bbr_cheating (133)

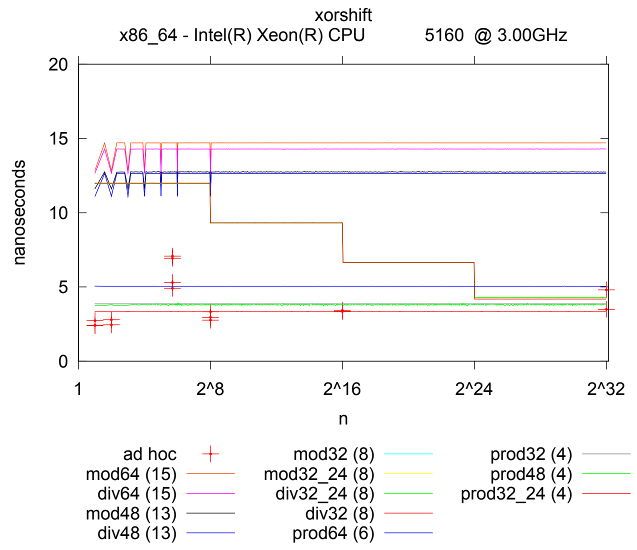
A.2.2 Integer arithmetic

```
typedef uint64_t st;
st div32(uint32_t n) {
  return my_gen_rand32() / n ;
}
st div32_24(uint32_t n) {
  return (my_gen_rand32() & 0xFF000000) / n ;
}
st div48(uint32_t n) {
  uint64_t r = my_gen_rand32(), m = r << 16;
  return m / n ;
}
st div64(uint32_t n) {
  uint64_t m = my_gen_rand64() / n ;
  return m;
}
st mod32(uint32_t n) {
  return my_gen_rand32() % n ;
}
st mod32_24(uint32_t n) {
  return (my_gen_rand32() & 0xFF000000) % n ;
}
st mod48(uint32_t n) {
  uint64_t r = my_gen_rand32(), m = r << 16;
  return m % n ;
}
st mod64(uint32_t n) {
  uint64_t m =my_gen_rand64() % n ;
  return m;
}
st prod32(uint32_t n) {
  return my_gen_rand32() * n ;
}
st prod32_24(uint32_t n) {
  return (my_gen_rand32() & 0xFF) * n ;
}
st prod48(uint32_t n) {
  uint64_t r = my_gen_rand32(), m = r << 16;
  return m * n ;
}
st prod64(uint32_t n) {
  uint64_t m = my_gen_rand64() * n ;
  return m;
}
```









B Code

The complete C code is available on request. The code that we wrote is licensed according to the *Gnu Public License v2.0.* (The *SFMT* code is licensed according to a modified BSD license — they are considered to be compatible).

In the following code, the macro COUNTBITS was used in testing efficiency; and was disabled while testing speeds.

B.1 Back-end RNGs

```
//

//to compile this code,  define RNG, by using  'gcc .... -DRNG=n', where n is
// 1  ->  SFMT , by M. Saito and M. Matsumoto
// 2  ->  SFMT + md5
// 3  ->  xorshift , by Marsaglia
// 4  ->  Blum Blum Shub with ~128bit (product of two ~31bit primes) (only on amd64, using gcc 128 int types)
// 5  ->  Blum Blum Shub with ~260bit modulus (product of two ~130bit primes)

//disclaimer: methods 2,4,5 are not guaranteed to generate high quality pseudonumbers; they were used
// only to test the code speed

/*************** SFMT ***/
#if 1==RNG
#ifdef HAVE_SSE2
char *RNGNAME="SFMT (sse)";
char *RNGNICK="sfmt_sse";
#else
char *RNGNAME="SFMT";
char *RNGNICK="sfmt";
#endif
#include "SFMT.h"

uint32_t my_gen_rand32()
{
  uint32_t r=gen_rand32();
  COUNTBITS(32);
  return r;
}

uint64_t my_gen_rand64()
{
  uint64_t r=gen_rand64();
  COUNTBITS(64);
  return r;
}

void my_init_gen_rand(uint32_t seed)
{
  init_gen_rand(seed);
}

/*************** SFMT + md5 ***/
#elif 2==RNG
#ifdef HAVE_SSE2
char *RNGNAME="SFMT (sse) + md5";
char *RNGNICK="sfmt_sse_md5";
#else
char *RNGNAME="SFMT + md5";
char *RNGNICK="sfmt_md5";
#endif
#include "SFMT.h"
#include "md5.h"

uint32_t my_gen_rand32()
{
  uint32_t I[8];
  for(int i=0;i<8;i++) I[i]=gen_rand32();
  unsigned char *UCp, output[16];
  UCp=(unsigned char*)I;
  md5(UCp, 32,  output);
  uint32_t  *U32p=(uint32_t *)output, r=*U32p;
  COUNTBITS(32);
  return r;
}

uint64_t my_gen_rand64()
{
  uint32_t  I[8];
  for(int i=0;i<8;i++) I[i]=gen_rand32();
  unsigned char *UCp, output[16];
  UCp=(unsigned char*)I;
  md5(UCp, 32,  output);
  uint64_t *U64p=(uint64_t *)output, r=*U64p;
  COUNTBITS(64);
  return r;
}

void my_init_gen_rand(uint32_t seed)
{
  if(md5_self_test(0)) exit(4);
  init_gen_rand(seed);
}

/***************  xorshift , by Marsaglia***/
#elif 3==RNG
char *RNGNAME="xorshift";
char *RNGNICK="xorshift";

static uint32_t __xorshift__x = 123456789;
static uint32_t __xorshift__y = 362436069;
static uint32_t __xorshift__z = 521288629;
static uint32_t __xorshift__w = 88675123;

uint32_t my_gen_rand32(void) {
  uint32_t t;
  t = __xorshift__x ^ (__xorshift__x << 11);
  __xorshift__x = __xorshift__y; __xorshift__y = __xorshift__z; __xorshift__z = __xorshift__w;
  COUNTBITS(32);
  return __xorshift__w = __xorshift__w ^ (__xorshift__w >> 19) ^ (t ^ (t >> 8));
}
uint64_t my_gen_rand64()
{
  uint64_t a=my_gen_rand32(), r = (a << 32) | my_gen_rand32();
  return r;
}
void my_init_gen_rand(uint32_t seed)
{
  __xorshift__x = 123456789 ^ seed;
  __xorshift__y = 362436069;
  __xorshift__z = 521288629;
  __xorshift__w = 88675123;
}

/******************* Blum Blum Shub (needs amd64 arch) ******************/
#elif 4==RNG
char *RNGNAME="Blum Blum Shub (64bit)";
char *RNGNICK="bbs64";

uint64_t my_seed=0x987fed5;

typedef __uint128_t uint128_t;

uint32_t my_gen_rand32(void) {
  const uint64_t p = 4222234259UL; //~ 2^31.9
  const uint64_t q = 4222231271UL;
  const uint64_t M =  p * q ;
  const uint64_t mask = 0xFFFFFFFFFFFFFFFFUL;
  uint128_t a = my_seed, b = a * a , my_seed = b % M;
  my_seed = c & mask ;
  COUNTBITS(32);
  return my_seed & 0xFFFFFFFFF;
}

uint64_t my_gen_rand64()
{
  uint64_t a=my_gen_rand32(), r = (a << 32) | my_gen_rand32();
  return r;
}

void my_init_gen_rand(uint32_t seed)
{
  my_seed=seed ^ 0x987fed5 ;
}

/***********************************************************/
/******************** Blum Blum Shub *******************/
#elif RNG==5
char *RNGNAME="Blum Blum Shub (260bit)";
char *RNGNICK="bbs260";
#include "gmp.h"

mpz_t bbs__pq__ ,  bbs__n__, bbs__64__, bbs ;
int initialized = 0;
void my_init_gen_rand(uint32_t seed)
{
  if(!initialized) {
    //http://primes.utm.edu/lists/small/small.html
    mpz_t p;    mpz_init_set_str (p, "361541588158511790855024350530978552 6231", 10);
    assert(mpz_probab_prime_p(p,12));
    mpz_t q;    mpz_init_set_str (q, "5570373270183181665098052481109678989411", 10);
    assert(mpz_probab_prime_p(q,12));
    mpz_mul(bbs__pq__ ,p ,q);
    mpz_clear(p); mpz_clear(q);
    mpz_init(bbs__n__);
    initialized=1;
  }
  unsigned long s=seed+0x100000000;
  mpz_set_ui(bbs__n__, s);
}

void bbs_step()
{
  mpz_t sqr; mpz_init(sqr);
  mpz_mul(sqr, bbs__n__, bbs__n__);
  mpz_tdiv_r(bbs__n__, sqr, bbs__pq__);
  mpz_clear(sqr);
}

uint32_t my_gen_rand32()
{
  bbs_step();
  unsigned long int r=mpz_get_ui(bbs__n__);
  COUNTBITS(32);
  if( sizeof(unsigned long int) == 4) {
    return r;
  } else {
    uint32_t rr=r & 0xFFFFFFFF;
    return rr;
  }
}

uint64_t my_gen_rand64()
{
  bbs_step();
  unsigned long int r=mpz_get_ui(bbs__n__);
  COUNTBITS(64);
  if ( sizeof(unsigned long int) == 8 ) {
    return r;
  } else {
    mpz_t q; mpz_init(q); mpz_tdiv_q_2exp (q, bbs__n__, 32);
    uint64_t r2=mpz_get_ui(q);
    mpz_clear(q);
    uint64_t rr=r | (r2 << 32);
    return rr;
  }
}

/***************  a counter , to test speeds ***/
#elif 11==RNG
char *RNGNAME="counter";
char *RNGNICK="counter";

static uint32_t my_seed32=0;
static uint64_t my_seed64=0;

uint32_t my_gen_rand32(void) {
  COUNTBITS(32);
  return my_seed32 += 0x4c1;
}

uint64_t my_gen_rand64()
{
  COUNTBITS(64);
  return my_seed64 += 0x4c7000004c1;
}

void my_init_gen_rand(uint32_t seed)
{
  my_seed32=seed;
  my_seed64=seed;
}

#else
#error "please define RNG"
#endif
//
```

B.2 *ad hoc* functions

```
//

unsigned int NextByte32() {
  static int l=0;
  static uint32_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand32();
    l=4;
  }
  unsigned int byte=R&255;
  l--;
  if(l) R>>=8;
  return byte;
}
unsigned int NextByte64() {
  static int l=0;
  static uint64_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand64();
    l=8;
  }
  unsigned int byte=R & 255;
  l--;
  if(l) R>>=8;
  return byte;
}
unsigned int __saved_bytes[8];
unsigned int NextByte64_prefilled() {
  static int l=0;
  if(unlikely(l<=0)) {
    uint64_t R=my_gen_rand64();
    __saved_bytes[0] = R & 255;
    R >>= 8;
    __saved_bytes[1] = R & 255;
    R >>= 8;
    __saved_bytes[2] = R & 255;
    R >>= 8;
    __saved_bytes[3] = R & 255;
    R >>= 8;
    __saved_bytes[4] = R & 255;
    R >>= 8;
    __saved_bytes[5] = R & 255;
    R >>= 8;
    __saved_bytes[6] = R & 255;
    R >>= 8;
    __saved_bytes[7] = R & 255;
    l=8;
  }
  l--;
  return __saved_bytes[l];
}
unsigned int NextWord32() {
  static int l=0;
  static uint32_t R=0;
  if(l<=0) {
    R=my_gen_rand32();
    l=2;
  }
  unsigned int bytes=R & 0xFFFF;
  R>>=16;
  l--;
  return bytes;
}
unsigned int NextWord64() {
  static int l=0;
  static uint64_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand64();
    l=4;
  }
  unsigned int bytes=R & 0xFFFF;
  R>>=16;
  l--;
  return bytes;
}
unsigned int Next2Bit32() {
  static int l=0;
  static uint32_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand32();
    l=16;
  }
  unsigned int bit=R&3;
  R>>=2;
  l--;
  return bit;
}
unsigned int Next2Bit64() {
  static int l=0;
  static uint64_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand64();
    l=32;
  }
  unsigned int bit=R&3;
  R>>=2;
  l--;
  return bit;
```

```
}
const static uint32_t bitmasks[32] = {
  0x1, 0x2, 0x4, 0x8 , 0x10, 0x20, 0x40, 0x80,
  0x100, 0x200, 0x400, 0x800, 0x1000, 0x2000, 0x4000, 0x8000,
  0x10000, 0x20000, 0x40000, 0x80000, 0x100000, 0x200000, 0x400000, 0x800000,
  0x1000000, 0x2000000, 0x4000000, 0x8000000, 0x10000000, 0x20000000, 0x40000000, 0x80000000
};
unsigned int NextBit32_by_mask() {
  static int l=0;
  static uint32_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand32();
    l=32;
  }
  l--;
  return (R & bitmasks[l]) ? 1 : 0;
}
unsigned int NextBit32() {
  static int l=0;
  static uint32_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand32();
    l=32;
  }
  unsigned int bit=R&1;
  R>>=1;
  l--;
  return bit;
}
unsigned int NextBit64() {
  static int l=0;
  static uint64_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand64();
    l=64;
  }
  unsigned int bit=R&1;
  R>>=1;
  l--;
  return bit;
}
//draws one card from a deck of 52 cards, using 32bit variables
unsigned int NextCard32() {
  static int l=0;
  static uint32_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand32();
    l=5;
  }
  l--;
  unsigned int c = R % 52;
  R /= 52;
  return c;
}
//draws one card from a deck of 52 cards, using 64 bit variables
unsigned int NextCard64()
{
  static int l=0;
  static uint64_t R=0;
  if(unlikely(l<=0)) {
    R=my_gen_rand64();
    l=11;
  }
  l--;
  unsigned int c = R % 52;
  R /= 52;
  return c;
}
//draws one card from a deck of 52 cards, using 32 bit variables
//and prefilling an array in memory
```

```
unsigned char __32saved_cards[5];
unsigned int NextCard32_prefilled() {
  static int l=0;
  if(unlikely(l<=0)) {
    uint32_t R=my_gen_rand32();  //52**5 < 2**32
    __32saved_cards[0] = R % 52;
    R /= 52;
    __32saved_cards[1] = R % 52;
    R /= 52;
    __32saved_cards[2] = R % 52;
    R /= 52;
    __32saved_cards[3] = R % 52;
    R /= 52;
    __32saved_cards[4] = R % 52;
    l=5;
  }
  l--;
  return __32saved_cards[l];
}
//draws one card from a deck of 52 cards, using 64 bit variables
//and prefilling an array in memory
unsigned char __64saved_cards[11];
unsigned int NextCard64_prefilled() {
  static int l=0;
  if(unlikely(l<=0)) {
    uint64_t R=my_gen_rand64();   //52**11 < 2**64
    __64saved_cards[0] = R % 52;
    R /= 52;
    __64saved_cards[1] = R % 52;
    R /= 52;
    __64saved_cards[2] = R % 52;
    R /= 52;
    __64saved_cards[3] = R % 52;
    R /= 52;
    __64saved_cards[4] = R % 52;
    R /= 52;
    __64saved_cards[5] = R % 52;
    R /= 52;
    __64saved_cards[6] = R % 52;
    R /= 52;
    __64saved_cards[7] = R % 52;
    R /= 52;
    __64saved_cards[8] = R % 52;
    R /= 52;
    __64saved_cards[9] = R % 52;
    R /= 52;
    __64saved_cards[10] = R % 52;
    l=11;
  }
  l--;
  return __64saved_cards[l];
} //
```

B.3 Uniform random generators

In the following code, the macro `COUNTFAILURES` was used in testing efficiency; and was disabled while testing speeds.

//

```
/**********  uniform_random_by_bit_recycling
         this implements the pseudocode in page 5
         (but pops 2 bits at a time)
*******************/
uint32_t uniform_random_by_bit_recycling(uint32_t n)
{
  static uint64_t m = 1, r = 0;
  const uint64_t N62=((uint64_t)1)<<62;
  while(1) {
    while(m<N62) { //fill the state
```

```
      r = (r<<2) | Next2Bit();
      m = m<<2;
    }
    const uint64_t q=m/n, nq = n * q;
    if( likely(r < nq) ) {
      uint32_t d = r % n; //remainder, is a random variable of modulus n
      r = r/n;            //quotient, is random variable of modulus q
      m = q;
      return  d;
    } else {
      COUNTFAILURES();
      m = m - nq;
      r = r - nq; // r is still a random variable of modulus m
    }
  }
}
/**********  uniform_random_by_bit_recycling
       this implements the pseudocode in page 5 ,
       but pops words,bytes,and pairs of bits
*******************/
uint32_t uniform_random_by_bit_recycling_faster(uint32_t n)
{
  static uint64_t m = 1, r = 0;
  const uint64_t N62=((uint64_t)1)<<62, N56=((uint64_t)1)<<56, N48=((uint64_t)1)<<48;
  while(1) {
    //fill the state
    if(m<N48) {
      r = (r<<16) | NextWord();
      m = m<<16;
    }
    while(m<N56) {
      r = (r<<8) | NextByte();
      m = m<<8;
    }
    while(m<N62) {
      r = (r<<2) | Next2Bit();
      m = m<<2;
    }
    const uint64_t q=m/n, nq=n*q;
    if( likely(r < nq) ) {
      uint32_t d = r % n; //remainder, is a random variable of modulus n
      r = r/n;            //quotient, is random variable of modulus q
      m = q;
      return  d;
    } else {
      COUNTFAILURES();
      m = m - nq;
      r = r - nq; // r is still a random variable of modulus m
    }
  }
}
/**********  uniform_random_by_bit_recycling_cheating
        this implements the pseudocode in page 6 ,
        but does not implement the "else" block, so it is not
        mathematically perfect; at the same time, since
        the probability of the "else" block would be less than 1/2^24
        the random numbers generated by this function are good enough
        for most purposes
*******************/
uint32_t uniform_random_by_bit_recycling_cheating(uint32_t n)
{
  static uint64_t m = 1, r = 0;
  const uint64_t N48=((uint64_t)1)<<48, N56=((uint64_t)1)<<56;
  if(m<N48) {   //fill the state
    r = (r<<16) | NextWord();
    m = m<<16;
  }
  while(m<N56) {
    r = (r<<8) | NextByte();
    m = m<<8;
  }
  uint32_t d = r % n;
  r = r/n;
  m = m/n;
  return  d;
}
/**********  uniform_random_by_bit_recycling32
      this implements the pseudocode in page5
      but only using 32bit variables, so it has some special
      methods when n>=2^29
******************/
uint32_t uniform_random_by_bit_recycling32(uint32_t n)
{
  const uint32_t N29=((uint32_t)1)<<29,  N30=((uint32_t)1)<<30,  N31=((uint32_t)1)<<31, N24=((uint32_t)1)<<24;
  //special methods
  if(n>=N31) {
    while(1) {
      uint32_t r = my_gen_rand32();
      if( likely(r < n) ) {
        return  r;
      } else {
        COUNTFAILURES();
      }
    }}
  if(n>=N30) {
    while(1) {
      uint32_t r = my_gen_rand32() >> 1;
      if( likely(r < n) ) {
        return  r;
      } else {
        COUNTFAILURES();
      }
    }}
  if(n>=N29) {
    while(1) {
      uint32_t r = my_gen_rand32() >> 2;
      if( likely(r < n) ) {
        return  r;
      } else {
        COUNTFAILURES();
      }
    }}
  //usual bit recycling
  static uint32_t m = 1, r = 0;
  while(1) {
    while(m<N24) { //fill the state
      r = (r<<8) | NextByte();
      m =  m<<8;
    }
    while(m<N30) { //fill the state
      r = (r<<2) | Next2Bit();
      m =  m<<2;
    }
    uint32_t q=m/n, nq=q*n;
    if(likely( r < nq) ) {
      uint32_t d = r % n; //remainder, is a random variable of modulus n
      r = r/n;            //quotient, is random variable of modulus q
      m = q;
      return d;
    } else {
      COUNTFAILURES();
      m = m - nq;
      r = r - nq; // r is still a random variable of modulus m
    }
  }
}
/**********  uniform_random_simple_64
         this is a simple implementation, found in many random number libraries
         this version uses 64 bit variables
******************/
```

```
uint32_t uniform_random_simple_64(uint32_t n)
{
  const uint64_t N=0xFFFFFFFFFFFFFFFFU; //2^64-1;
  uint64_t q = N/n, nq=((uint64_t)n) * q;
  while(1) {
    uint64_t r = my_gen_rand64();
    if( likely(r < nq) ) {
      uint32_t d = r % n; //remainder, is a random variable of modulus n
      return  d;
    } else {
      COUNTFAILURES();
    }
  }
}
/**********  uniform_random_simple
         this is a simple implementation, found in many random number libraries
         this version uses 32 bit variables
*******************/
uint32_t uniform_random_simple_32(uint32_t n)
{
  const uint32_t N=0xFFFFFFFFU; // 2^32-1;
  uint32_t q = N/n;
  while(1) {
    uint32_t r = my_gen_rand32();
    if( likely(r < n * q)) {
      uint32_t d = r % n; //remainder, is a random variable of modulus n
      return  d;
    } else {
      COUNTFAILURES();
    }
  }
}
/**********  uniform_random_simple_recycler
         this is a simple implementation, with recycling for small n
         this version uses 32 bit variables
*******************/

uint32_t uniform_random_simple_recycler(uint32_t n)
{
  static  uint32_t _r=0, _m=0;
  const uint32_t N=0xFFFFFFFFU; //2^32-1
  if (_m>n) {
      uint32_t d = _r % n;
      _m/=n;
      _r/=n;
      return d;
    }
  const uint32_t q = N/n, nq=n*q;
  while(1) {
    uint32_t newr = my_gen_rand32();
    if(newr < nq) {
      uint32_t d = newr % n; //newr is a random variable of modulus n
      if(_m<q) { //there is more entropy in newr than in _r
        _m=q;
        _r=newr/n;
      }
      return  d;
    } else {
      COUNTFAILURES();
    }
  }
}
/**********  uniform_random_simple
     some alternative versions, using 64 bit variables
*******************/
uint32_t uniform_random_simple_40(uint32_t n)
{
  const uint64_t N=((uint64_t)1)<<40;
  uint64_t q = N/n, nq=((uint64_t)n) * q;
  while(1) {
    uint64_t r = my_gen_rand32();
    r=(r<<8) | NextByte(); //create 40 bits random numbers
    if( likely(r < nq) ) {
      uint32_t d = r % n; //remainder, is a random variable of modulus n
      return  d;
    } else {
      COUNTFAILURES();
    }
  }
}

uint32_t uniform_random_simple_48(uint32_t n)
{
  const uint64_t N=((uint64_t)1)<<48;
  uint64_t q = N/n, nq=((uint64_t)n) * q;
  while(1) {
    uint64_t r = my_gen_rand32();
    r=(r<<16) | NextWord(); //create 48 bits random numbers
    if( likely(r < nq) ) {
      uint32_t d = r % n; //remainder, is a random variable of modulus n
      return  d;
    } else {
      COUNTFAILURES();
    }
  }
}
//
```

Acknowledgments

The author thanks Professors S. Marmi and A. Profeti for allowing access to hardware.

Bibliography

References

[1] “Doctor Jacques” at the Math forum http://mathforum.org/library/drmath/view/65653.html (2004)

[2] Mutsuo Saito and Makoto Matsumoto, *SIMD oriented Fast Mersenne Twister(SFMT): a 128-bit Pseudorandom Number Generator* Monte Carlo and Quasi-Monte Carlo Methods 2006, Springer, 2008, pp. 607 – 622. DOI 10.1007/978-3-540-74496-2_36. *Code (ver. 1.3.3) available from* http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html *(See Mutsuo Saito's Master's Thesis for more detailed information).*

[3] Mutsuo Saito and Makoto Matsumoto, *"A PRNG Specialized in Double Precision Floating Point Number Using an Affine Transition"*, Monte Carlo and Quasi-Monte Carlo Methods 2008, Springer, 2009, pp. 589 – 602. DOI 10.1007/978-3-642-04107-5_38

[4] George Marsaglia *Xorshift RNGs* Journal of Statistical Software, 8, 1-9 (2003). http://www.jstatsoft.org/v08/i14/.

[5] L. Blum, M. Blum, and M. Shub, *A Simple Unpredictable Pseudo-Random Number Generator* SIAM J. Comput. 15, 364 (1986), DOI 10.1137/0215025