EFFICIENT REVERSIBLE UNIFORM AND NON-UNIFORM RANDOM NUMBER GENERATION IN UNU.RAN

Srikanth B. Yoginath and Kalyan S. Perumalla
Computer Science and Mathematics Division
Oak Ridge National Laboratory*
Oak Ridge, TN, USA.
yoginathsb@ornl.gov, perumallaks@ornl.gov

ABSTRACT

Reversible random number generations are useful in large-scale fault-tolerant parallel computations and parallel discrete event simulations that are based on reversible computation. The Universal Non-Uniform Random Number Generator (UNU.RAN) is one of the popular random number generators used in the simulation community, but the generators are forward-only in nature. In this paper, we develop new reverse algorithm for the default uniform random number generator algorithm of UNU.RAN and also a few non-uniform random generators that use the Transform Density Reduction (TDR) method. We verify the correctness of reversals of our algorithms and also provide performance results to demonstrate reverse computing runtime adds little overheads relative to its forward counterpart.

Keywords: Reverse-computing, Reverse random number generator

1 INTRODUCTION

Bi-directional (forward and reverse) execution finds use in fault-tolerant high performance computing, rollback-based optimistic parallel simulations, large-scale debugging, and other areas (Perumalla 2013, Carothers et al. 1999, Bishop 1997, Boothe et al. 2000, Lee et al. 2002, Tang et al. 2006). However, bi-directional execution by log-based reversal typically incurs high memory costs. Probability distribution sampling, used in many scientific codes, is one of the challenging components to reverse, due to the large amount of memory needed to store long traces. Naïve approaches can render bi-directional execution impractical due to high memory costs of the execution that the traces need to achieve reversibility (Lin et al. 1990, Palaniswamy et al. 1993, Cleary et al. 1994, West et al. 1996, Gomes et al. 1996). This problem is especially pronounced in high performance computing, due to the dramatic increases of trace lengths that result from very high speed of execution. For example, a random number generator can be thrown a million times in a second, whose reversal by log-based methods can require several megabytes of memory for each generator. The memory cost is even more amplified when the quality of the random number generator is increased by increasing the seed size. Thus, for enabling bi-directional computation, new memory-efficient schemes must be developed that either minimize the overhead, or, ideally, eliminate the memory needed for reversal. Here we focus on one of the core operations that occurs in scientific simulations, namely, sampling of complex probability distributions, and provide a new scheme that eliminates the memory cost for reversal.

Sampling of probability distributions is routinely employed in computer models of physical systems. Pseudo-random number streams are used to generate samples that conform to the desired probability distributions. In computationally intensive simulations, a large number (millions to billions) of samples are

*This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

drawn. For simple distributions, the sampling procedure is given by a closed-form formula for certain distributions (such as the Exponential distribution). In more complex distributions, the sampling formula is either computationally complex, or, worse, may not be expressible as a closed-form expression. In such complex distributions, a different sampling procedure is employed, using an iterative approach to progressively approach the desired distribution.

In the case of simple probability distributions, such as Exponential or Pareto distributions, closed-form inversions of their cumulative distribution functions (CDF) are known. For such distributions, it has been shown that perfect reversibility can be achieved (Carothers et al. 1999, Perumalla et al. 1998). This is realized by employing reversible uniform random number generators. Reversing the sampling operation on an exponential distribution, thus, becomes as simple as invoking the reversal of the underlying uniform random number generator once per reversal step. The restoration of the uniform random number seed is necessary and sufficient for reversing the sampling of the distribution.

However, for more complex distributions for which either the inverse of the CDF is computationally prohibitive or a closed form representation of the inverse of the CDF is unknown, a simple reversal based on restoration of random number seeds does not work. This is because of two reasons: (1) the unavailability of closed-form inversion of CDF, or its prohibitive computational complexity for exact computation, gives rise to iterative code that uses multiple uniform random number samples, and (2) the number of uniform samples used up for any given non-uniform sample varies with each sample, and hence cannot be easily reversed. For example, in First Passage Time (FPT) distributions (Redner 2001, Perumalla and Donev 2009), the sampling formulas are computationally intensive and difficult to express as closed form expressions. In such complex distributions, a different sampling procedure is employed, using an iterative approach to progressively approach the desired distribution.

The inversion challenge for such distributions is rooted in the fact that control flow is infused into the sampling procedure. Such control flow information is absent in sampling simple distributions. When control flow complexity is introduced into this method, the one-to-one correspondence between the random number seed stream and the probability distribution sample stream gets broken. Thus, reversal of a sample s_i might require an unknown number, $n_i > 0$, of reversals of updates to the underlying random number seed. Since each sample s_i has a value n_i that may be different from other samples, a log is *apparently* needed to keep track of the number of iterations performed for each sample in the forward direction. Thus, the log is: (a) proportional to the number of samples, and (b) theoretically unbounded in the amount of memory needed to remember each sample's iteration count. In practice, the theoretically unbounded nature of control flow information for *each* sample can be capped with a sufficiently large integer variable. Nevertheless, the proportionality of the trace size with the sample stream length (number of samples drawn) is the most dominant factor on memory. It is this trace length that we reduce (in fact, eliminate) with our reversal procedure.

We solve the problem of minimizing the memory needed to move forward as well as backward, at will, in a stream of samples generated by a rejection-based procedure used to generate samples from probability distributions. Our scheme completely eliminates the memory overhead. The scheme is perfectly reversible in the sense that the memory needed to go backwards is independent of the sampled stream length. Our solution provides two important features needed for use in parallel programs, namely, determinism and repeatability, across arbitrarily spaced changes of direction.

While these algorithms and methodologies have been documented earlier (Perumalla 2013), they have not been applied to any popular, production-level random number generator software and hence their utilization is limited. In this paper, we take a step towards bridging this gap by developing a methodology using which the reverse-computing abilities can be brought into the folds of popular RNG packages. For this purpose, we use the Universal Non-Uniform Random Number Generator (UNU.RAN). The UNU.RAN library is a collection of algorithms for generating non-uniform pseudo-random number variates and is considered suitable in almost all situations for experimenting with different distributions including non-standard distributions. In the following section, we introduce few methods of UNU.RAN package for uniform and non-uniform random number generation. The reversal algorithms and implementations of the forward random number generation routines discussed in section 3 is discussed in section 3. In section 4, we present the verification and performance results, and finally conclude in section 5.

2 FORWARD COMPUTING IN UNU.RAN

2.1 UNU.RAN Uniform RNG

In default settings, UNU.RAN package uses a fast combined multiple recursive generators (L'Ecuyer 2000, L'Ecuyer 2012) for generating uniformly distributed random number generation. This uniform RNG is used for generating random numbers for any given probability density functions (PDF). Hence, it becomes essential to reverse the uniform RNG at the core of the UNU.RAN package. Further, for effective usability, reversal speed must be comparable to its forward counter-part. The MRG31k3p version of combined Multiple Recursive Generators (MRG) (L'Ecuyer 2000) is used as the default uniform RNG in UNU.RAN.

$$x_{1,n} = \left(a_{11}x_{1,n-1} + a_{12}x_{1,n-2} + a_{13}x_{1,n-3}\right) \, mod \, m_1 \quad (1)$$

$$x_{2,n} = \left(a_{21}x_{2,n-1} + a_{22}x_{2,n-2} + a_{23}x_{2,n-3}\right) \, mod \, m_2 \quad (2)$$

A random number is generated as $\frac{(x_{1,n} - x_{2,n}) \, mod \, m_1}{m_1}$ (3)

```
# define m1       2147483647
# define m2       2147462579
# define norm     4.656612873077393e-10
# define mask11   511
# define mask12   16777215
# define mask20   65535
double unur_urng_MRG31k3p (void *dummy )
/* Combined multiple recursive generator. */
/* Copyright (c) 2002 Renee Touzin.       */
{
  register unsigned long yy1, yy2;  /* For intermediate results */
  /* First component */
  yy1 = ( (((x11 & mask11) << 22) + (x11 >> 9))
      + (((x12 & mask12) << 7)  + (x12 >> 24)) );
  if (yy1 > m1) yy1 -= m1;
  yy1 += x12;
  if (yy1 > m1) yy1 -= m1;
  x12 = x11;  x11 = x10;  x10 = yy1;
  /* Second component */
  yy1 = ((x20 & mask20) << 15) + 21069 * (x20 >> 16);
  if (yy1 > m2) yy1 -= m2;
  yy2 = ((x22 & mask20) << 15) + 21069 * (x22 >> 16);
  if (yy2 > m2) yy2 -= m2;
  yy2 += x22;
  if (yy2 > m2) yy2 -= m2;
  yy2 += yy1;
  if (yy2 > m2) yy2 -= m2;
  x22 = x21;  x21 = x20;  x20 = yy2;
  /* Combination */
  if (x10 <= x20)
    return ((x10 - x20 + m1) * norm);
  else
    return ((x10 - x20) * norm);
} /* end of unur_urng_MRG31k3p() */
```

Figure 1 Combined MRG with powers of 2 decomposition method (MRG131k3p implementation in UNU.RAN)

The source code of MRG31k3p implemented in C language is shown in Figure 1. This combined MRG has $J = 2$ components of order $k = 3$. The two components are defined by parameters $m_1 = 2^{31} - 1$, $a_{11} = 0$, $a_{12} = 2^{22}$, $a_{13} = 2^7 + 1$, $m_2 = 2^{31} - 21069$, $a_{21} = 2^{15}$, $a_{22} = 0$, $a_{23} = 2^{15} + 1$. In the code shown in Figure 1, the RNG starts with six seed variables $x_{10}, x_{11}, x_{12}, x_{20}, x_{21}, x_{22}$ which correspond to $x_{1,n-1}, x_{1,n-2}, x_{1,n-3}, x_{2,n-1}, x_{2,n-1}, x_{2,n-2}, x_{2,n-3}$ of the above equation. At the end of each generation, x_{10} takes the newly generated random number, while the x_{11} and x_{12} take the values of x_{10} and x_{11}, respectively. Similarly, x_{20} takes the newly generated random number, while x_{21} and x_{22} are overwritten by x_{20} and x_{21} values respectively, as indicated in the equations (1) and (2).

2.2 UNU.RAN Non-uniform RNG

UNU.RAN implements several methods for generating random numbers. However, the choice is dependent on the type and amount of information about the required distribution that can be provided and the familiarity of the user about different methods. Several different methods for generating non-uniform random variates have been developed and are discussed in UNU.RAN User Manual. However, in this paper, we focus on the "Transformed Density Rejection" (TDR) method and experiment with univariate continuous distribution to demonstrate our perfect reversal capabilities. The methodologies developed here are equally applicable to other supported UNU.RAN methods of RNG generation. The TDR is an acceptance/rejection method that uses the concavity of a transformed density to construct the hat and squeeze functions automatically [5].

The function call *unur_tdr_ps_sample* is used for the non-uniform random number generation with the TDR method. Based on acceptance-rejection, this method samples random points that are uniformly distributed and checks whether the points fall within the density curve of a given distribution. If the randomly picked point falls within the density function then the point is accepted; if not, it is discarded, and the same procedure is repeated with newly sampled set of points. This works for any input distribution with bounded density on a bounded domain. The TDR uses the *hat* and *squeeze* functions for efficiently generating the random numbers for the input density function. The following is the general procedure followed by TDR method.

```
1. Generate a $U(0,1)$ random number U
2. Set $X$ to $H^{-1}(U)$
3. Generate $U(0,1)$ random number $V$
4. Set $Y$ to $Vh(X)$
5. If ($Y \le f(X)$ || $Y \le s(X)$) accept $X$ as the random variate
6. Else try again (goto 1)
```

Figure 2 Pseudocode of the forward procedure *unur_tdr_ps_sample*

In the above procedure, U(0,1) samples random numbers between 0 and 1 from an uniform RNG. The H^{-1} function corresponds to the CDF of the hat function with density h(X), which is chosen such that h(X) ≥ f(X) for all X in the domain of given distribution whose density function is f(X). To generate a (X, Y) pair, X is generated from a distribution with density proportional to h(X) and Y is a random number from uniform RNG from 0 and h(X). If Y falls within the density of the given distribution, then X is accepted, else rejected. Since, computing f(X) is expensive or time-consuming, a simpler lower-bound squeeze function s(X) is used for quick acceptance, that is if Y ≤ s(X), X is accepted (see UNU.RAN User Manual).

3 REVERSE COMPUTING IN UNU.RAN

3.1 Reverse UNU.RAN Uniform RNG

In this reverse MRG31k3p implementation shown in Figure 3, four of the six values of $x_{i,j}$ of the combined MRG can be reversed, as these values are copied over on to other variables. That is, values held by $x_{11}, x_{12}, x_{21}, x_{22}$ in the current cycle are $x_{10}, x_{11}, x_{21}, x_{22}$ of the previous cycle. Further, the values held by current x_{10} and x_{20} correspond to the values obtained by computing the two MRG equations. With this observation as a starting point, we need to recover the values of x_{12} and x_{22} that were overwritten in the previous cycle of the random number generation. Let yy_1 and yy_2 be the previous result from the first and second MRG equations, respectively. Note that yy_1 has components of x_{11} and x_{12}, while yy_2 has components corresponding to x_{20} and x_{22}. First, from the yy_1 and yy_2, the values corresponding to components x_{11} and x_{20} are negated, respectively. This results in two simpler *multiplicative linear congruential generator* (MLCG) equations, where $x_{i,n+1}$ and the multiplier a_i are known and $x_{i,n}$ needs to be determined in $x_{i,n+1} = \left(a_i x_{i,n}\right) \, mod \, m_i$. Obtaining efficient reversals from such equations has been addressed previously by computing $b_i = \left(a_i^{m-2} x_{i,n}\right) \, mod \, m_i$, and $x_{i,n}$ is reverse-computed using b_i as $x_{i,n} = \left(b_i x_{i,n+1}\right) \, mod \, m_i$ (Carothers et al. 1999).

3.2 Reverse UNU.RAN Non-uniform RNG Procedure

The complexity in the reverse algorithm for this procedure arises from the fact that the acceptance of a random variate does not use a constant number of steps and it can be only determined at the runtime. This is because a random variate can be accepted or rejected based on the actual set of random numbers generated and on the input density function. Hence, for every step in reversal, we need to ensure that the random variate was accepted and not rejected. The following is the procedure for reverse TDR at every step of reversal.

```
double rev_unur_urng_MRG31k3p (void *dummy )
{
  long yy1, yy2;
  if (firstFlag == 0){
    b22 = FindB(a23, m2);
    b12 = FindB(a13, m1);
    firstFlag = 1;
  }
  yy2 = x20;
  yy1 = x10;
  // reverse unchanged values
  x10 = x11;
  x11 = x12;
  x20 = x21;
  x21 = x22;
  // find values of x12 and x22 from yy1 and yy2
  // find x12
  long tx12n = (((x11 & mask11) << 22) + (x11 >> 9));
  if (tx12n > m1) tx12n -= m1;
  long tx12 = yy1 - tx12n;
  if (tx12 < 0){tx12 += m1;}
  x12 = (b12*tx12)%m1;
  //find x22
  long tx22n = (((x20 & mask20) << 15) + 21069 * (x20 >> 16));
  if (tx22n > m2)  tx22n -= m2;
  long tx22 = yy2 - tx22n;
  if(tx22 < 0 ) {tx22 += m2;}
  x22 = (b22*tx22)%m2;
}
```

Figure 3 Code of the reverse procedure of MRG31k3p

```
1. R(U(0,1)), R(U(0,1))
2. Generate a U(0,1) random number U
3. Set X to H^-1(U)
4. Generate U(0,1) random number V
5. Set Y to Vh(X)
6. If (Y ≤ f(X) || Y ≤ s(X) ) R(U(0,1)), R(U(0,1)) break;
8. Else R(U(0,1)), R(U(0,1)) and try again (goto 1)
```

Figure 4 Pseudocode of reverse *unur_tdr_ps_sample*

To start the reversal, we reverse the uniform RNG twice (*R(U(0,1))* in line 1 of Figure 4) since we know that two uniform random numbers have been certainly used for a successful TDR accept. The following procedure from (2 to 5), essentially checks if the set of uniform random numbers results in an acceptance; if yes, it reverts back to the uniform RNGs it started with, thus completing the single-step reversal. If the result from the uniform random numbers was rejected, then the reversal procedure requests additional uniform RNG rollbacks and continues this procedure until the generated RNGs are accepted. Note that the very first reverse-TDR after the last forward TDR will reverse back in a single step. However, this is not essentially true for multi-step reversals or in between two consecutive reversals.

As seen in Figure 5, the reverse computing *_unur_tdr_ps_rev_sample* is analogous to its forward counterpart (*_unur_tdr_ps_sample*), with a few additional uniform random number reversal calls (highlighted in Figure 5). For testing purpose, we provide access to the uniform random number reversal call (*rev_unur_urng_MRG31k3p*) through the *anti* variable of UNU.RAN generator (*struct unur_gen*). The number of reversal steps to be performed is provided by the user as an input through the *nRev* argument.

```
double _unur_tdr_ps_rev_sample( struct unur_gen *gen, int nRev)
{
  UNUR_URNG *urng;         /* pointer to uniform random number generator */
  struct unur_tdr_interval *iv;
  double U, V;                /* uniform random number                         */
  double X;                   /* generated point                               */
  double fx;                  /* value of density at X                         */
  double Thx;                 /* value of transformed hat at X                 */
  int i = 0;
  for (i=0; i<nRev; i++)
  {
    /* main URNG */
    urng = gen->urng;
    while (1)
    {
      (urng)->anti((urng)->state, 0);
      (urng)->anti((urng)->state, 0);
      /* sample from U( Umin, Umax ) */
      U = GEN->Umin + _unur_call_urng(urng) * (GEN->Umax - GEN->Umin);
         :
         :
      /* accept or reject */
      V = _unur_call_urng(urng);
      /* squeeze rejection */
      if(V <= iv->sq) break; //returning in squeeze rejection;
         :
         :
      /* evaluate PDF at X */
      fx = PDF(X);
      /* main rejection */
      if (V <= fx) break; //returning in main rejection
      (urng)->anti((urng)->state, 0);
      (urng)->anti((urng)->state, 0);
    }
    (urng)->anti((urng)->state, 0);
    (urng)->anti((urng)->state, 0);
  }
}
```

Figure 5 Code excerpt of UNU.RAN TDR sampling reversal (*_unur_tdr_ps_rev_sample*)

4 VERIFICATION AND PERFORMANCE EVALUATION

The experiments were conducted on a Linux System with 6-core AMD Phenom(tm) II X6 1100T processor, with 16GB of memory. UNU.RAN library version 1.8.1 was used in our experimentations.

4.1 Verification of UNU.RAN uniform RNG Reversal

To verify the exactness of reversals, we generate *n* random numbers followed by *n* reversals, ensuring that, at each step, the reversal produces the previously generated random number. To visually convey the correctness of the reversals, we generate several millions of random numbers, sample every 50000th number and plot the obtained random numbers for both forward and reverse cases. The reverse plot would start where the forward plot stopped and, for a correct reversal, the plot would exactly retrace the path of forward plot in the reverse order. Figure 6 plots the forward and reverse samples from the MRG31k3p uniform random number generator, which visually conveys a perfect reversal. Most of the experimentations are performed on the example sets made available by UNU.RAN. Note that we verified the numbers programmatically for exact match, and this plot is only to convey its operation visually.

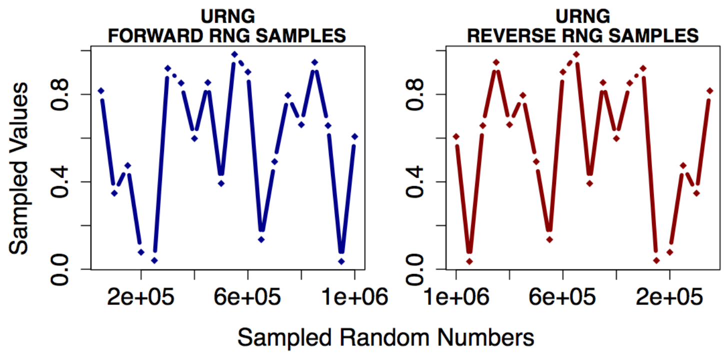

Figure 6 Forward and reverse sampling of every 50000th number of the generated million uniformly distributed random numbers

4.2 Verification of UNU.RAN non-uniform RNG Reversal

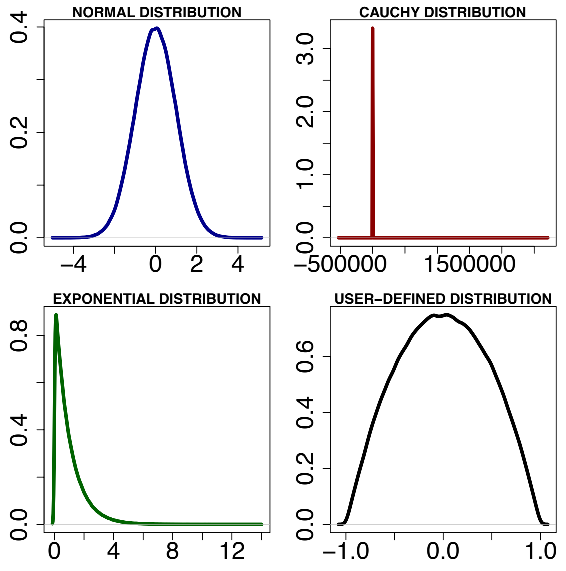

Figure 7 Density plot from the random numbers (million each) generated from various UNU.RAN functions using TDR method

Using UNU.RAN for sampling from a particular distribution involves following steps (a) creation of a distribution object (b) choosing a method for RNG (TDR in our case) (c) initializing the generator (d) using the generator object to generate random numbers. With the TDR method, we can generate random numbers corresponding to well-known distributions such as the normal distribution or we can also generate random numbers based on user-defined distributions by providing the function pointers to the PDF function and its derivative function. In our experimentation with non-uniform RNG, we have used three standard distributions, namely, Normal, Cauchy's and Exponential distributions, and one user defined distribution for which the PDF function and its derivative are as shown in Figure 8. The PDFs of all the distributions for which non-uniform RNGs were generated are plotted in Figure 7. For non-uniform RNG using

UNU.RAN we chose four distributions for verification. The forward and reverse plots for the non-uniform RNG verification were generated using same procedure used for uniform RNG as discussed in the previous section. The verification plots for normal, Cauchy, exponential and user-defined are shown in Figure 9. A perfect reversal is observed in all these plots.

```
/* ------------------------------------------------------------- */
/* Define the PDF and dPDF of our distribution.                  */
/*                                                               */
/* Our distribution has the PDF                                  */
/*                                                               */
/*         /  1 - x*x  if |x| <= 1                               */
/*  f(x) = <                                                     */
/*         \  0        otherwise                                 */
/*                                                               */
/* The PDF of our distribution:                                  */
double mypdf( double x, const UNUR_DISTR *distr )
{
   if (fabs(x) >= 1.)
     return 0.;
   else
    return (1.-(x*x));
} /* end of mypdf() */

/* The derivative of the PDF of our distribution:                */
double mydpdf( double x, const UNUR_DISTR *distr )
{
   if (fabs(x) >= 1.)
     return 0.;
   else
     return (-2.*x);
} /* end of mydpdf() */
```

Figure 8 PDF and dPDF of the user-defined distribution

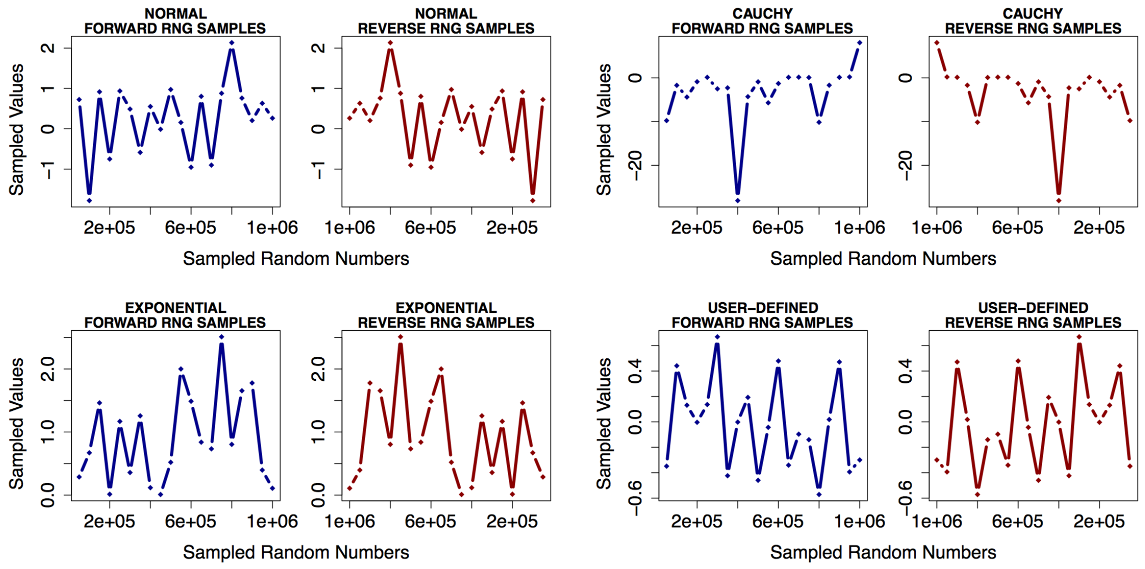

Figure 9 Forward and reverse sampling of every 50000th number of the generated million random numbers from normal, Cauchy, exponential and user-defined distributions

In Figure 10, we plot the runtimes (in microseconds) for forward and reverse procedures for uniform random numbers on the left chart and all the non-uniform random numbers on the right chart. Nearly identical runtimes are observed for the forward and reversal procedures in uniform RNG. For non-uniform RNGs, the runtimes for all the non-uniform test distributions have their reversal runtimes slightly higher than their forward counterpart. Cauchy forward and reversal runtimes are slightly higher than other distributions, perhaps because of the skinny PDF, which might result in many non-accepted random numbers and, which also impacts the runtime of the reversals similarly.

4.3 Runtime Performance

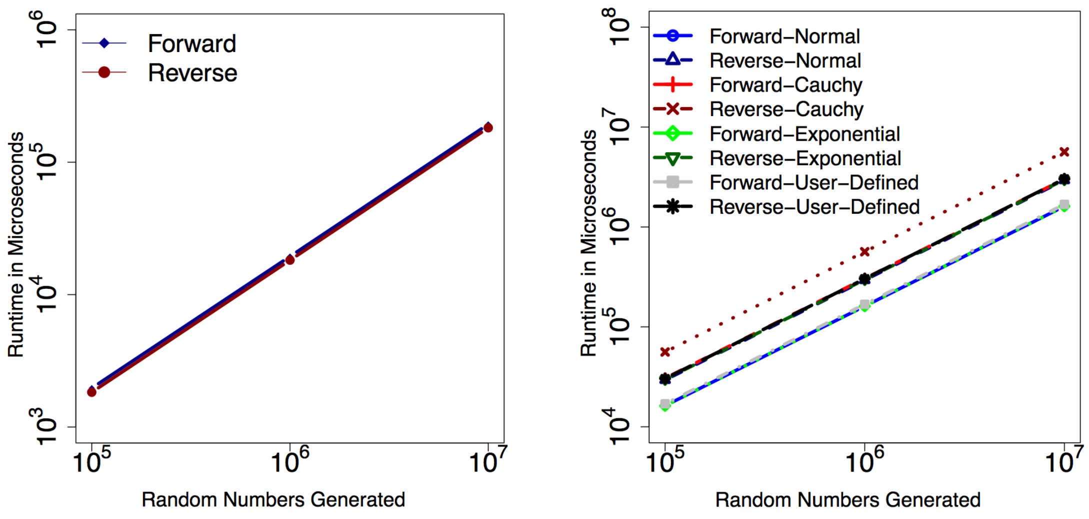

Figure 10 Comparison of mean execution runtimes (from 100 sample runs) of forward and reverse random number generations. Uniform random numbers(left) and Non-uniform random numbers (right)

5 SUMMARY AND FUTURE WORK

In this paper, we addressed how reverse computations can be made available in well-known packages like, UNU.RAN. We developed an efficient algorithm to reverse the default uniform random number generator (MRG31k3p). Then we provided methodology and implementation specifics on infusing reverse computing capabilities for non-uniform RNG using TDR approach. In each case, we demonstrated the effectiveness and efficiency through the verification and runtime performance results. We also demonstrated the possibility reversals without much change to the original code structure of UNU.RAN. In our future work we intend to broaden the support for reverse computations within the UNU.RAN software framework.

REFERENCES

P. L'Ecuyer and R. Touzin. Fast combined multiple recursive generators with multipliers of the form a=±2 q±2 r. Proceedings of the 32nd conference on Winter simulation. Society for Computer Simulation International, 2000.

P. L'Ecuyer. Random number generation. Springer Berlin Heidelberg, 2012.

C. Carothers, K. S. Perumalla, and R. M. Fujimoto, Efficient Optimistic Parallel Simulations using Reverse Computation. ACM Transactions on Modeling and Computer Simulation, 1999. 9(3): p. 224–253.

UNU.RAN User Manual http://statmath.wu.ac.at/unuran/doc/unuran.html

K.S. Perumalla. Introduction to reversible computing. CRC Press, 2013.

W. Hörmann. A rejection technique for sampling from T-concave distributions. ACM Transactions on Mathematical Software (TOMS) 21.2 (1995): 182-193.

P. Bishop, Using Reversible Computing to Achieve Fail-Safety. in ISSRE-97. 1997. IEEE Computer Society Press.

B. Boothe, Efficient Algorithms for Bidirectional Debugging. in Programming Language Design and Implementation. 2000. ACM Press.

J. Lee, et al., Reversible Computation in Asynchronous Cellular Automata. Lecture Notes in Computer Science, 2002. 2509: p. 220–229.

Y. Tang, et al., Optimistic Simulations of Physical Systems using Reverse Computation. SIMULATION: Transactions of The Society for Modeling and Simulation International, 2006. 82(1): p. 61–73.

Y.B. Lin and E. D. Lazowska, Reducing the State Saving Overhead for Time Warp Parallel Simulation. 1990, Computer Science Department, University of Washington: Seattle, Washington.

A. C. Palaniswamy and P. A. Wilsey, An Analytical Comparison of Periodic Checkpointing and Incremental State Saving, in Proceedings of the 7th Workshop on Parallel and Distributed Simulation. 1993. p. 127–134.

J. Cleary, et al., Cost of State Saving and Rollback, in Proceedings of the 8th Workshop on Parallel and Distributed Simulation. 1994. p. 94–101.

D. West and K. Panesar, Automatic Incremental State Saving, in Proceedings of the 10th Workshop on Parallel and Distributed Simulation. 1996. p. 78–85.

F. Gomes, Compiler Techniques for State Saving in Parallel Discrete Event Simulation, in Computer Science. 1996, University of Calgary, Canada.

K. Perumalla, R. Fujimoto, and A. Ogielski, TeD - A Language for Modeling Telecommunications Networks. Performance Evaluation Review, 1998. 25(4).

K. S. Perumalla and R. M. Fujimoto, Source Code Transformations for Efficient Reversibility. 1999, College of Computing, Georgia Institute of Technology, Atlanta.

S. Redner, A Guide to First-Passage Processes. 2001: Cambridge University Press

K. Perumalla and Alexandar Donev, Perfect Reversal of Rejection Sampling Methods for First-Passage-Time and Similar Probability Distributions. 2009. Technical Memorandum ORNL/TM-2009/182, Oak Ridge National Laboratory, Oak Ridge.

R. M. Fujimoto, Optimistic Approaches to Parallel Discrete Event Simulation. Transactions of the Society for Computer Simulation, 1990. 7(2): p. 153–191.

K. S. Perumalla, µsik - A Micro-Kernel for Parallel/Distributed Simulation Systems. in Workshop on Principles of Advanced and Distributed Simulation. 2005.

AUTHOR BIOGRAPHIES

Srikanth B. Yoginath is a research staff member in the Computer Science and Mathematics Division of the Oak Ridge National Laboratory. He holds a PhD in Computational Sciences and Engineering from Georgia Institute of Technology. His email address is yoginathsb@ornl.gov.

Kalyan S. Perumalla is a Distinguished Research Staff Member and manager in the Computer Science and Mathematics Division at the Oak Ridge National Laboratory, USA, where he leads the Discrete Computing Systems Group. He also serves as an Adjunct Professor in the School of Computational Sciences and Engineering, Georgia Institute of Technology, USA. His e-mail address is perumallaks@ornl.gov.