OAK RIDGE
NATIONAL LABORATORY

MANAGED BY UT-BATTELLE
FOR THE DEPARTMENT OF ENERGY

ORNL/TM-2009/182

Perfect Reversal of Rejection Sampling Methods for First-Passage-Time and Similar Probability Distributions

August 2009

Prepared by
Kalyan S. Perumalla
Senior Research Staff Member

UT-BATTELLE

ORNL-27 (4-00)

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via the U.S. Department of Energy (DOE) Information Bridge.

Web site http://www.osti.gov/bridge

Reports produced before January 1, 1996, may be purchased by members of the public from the following source.

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone 703-605-6000 (1-800-553-6847)
TDD 703-487-4639
Fax 703-605-6900
E-mail info@ntis.gov
Web site http://www.ntis.gov/support/ordernowabout.htm

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange (ETDE) representatives, and International Nuclear Information System (INIS) representatives from the following source.

Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Telephone 865-576-8401
Fax 865-576-5728
E-mail reports@osti.gov
Web site http://www.osti.gov/contact.html

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

ORNL/TM-2009/182

Computational Sciences and Engineering Division

PERFECT REVERSAL OF REJECTION SAMPLING METHODS FOR FIRST-PASSAGE-TIME AND SIMILAR PROBABILITY DISTRIBUTIONS

Kalyan S. Perumalla†
Aleksandar Donev‡

Date Published: August 2009

Prepared by
OAK RIDGE NATIONAL LABORATORY
Oak Ridge, Tennessee 37831-6283
Managed by
UT-BATTELLE, LLC
for the
U.S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

† Oak Ridge National Laboratory
‡ Lawrence Livermore National Laboratory

CONTENTS

Page

LIST OF FIGURES iv
ABSTRACT v
1. INTRODUCTION 1
  1.1 BI-DIRECTIONAL EXECUTION 1
  1.2 SAMPLING PROBABILITY DISTRIBUTIONS 1
  1.3 REVERSIBLE RANDOM NUMBER GENERATORS 2
  1.4 REJECTION SAMPLING METHOD 2
2. FORWARD FORMULATION 3
  2.1 INVOCATION 3
  2.2 SOURCE CODE 3
3. REVERSAL 5
  3.1 OBSERVATIONS 5
  3.2 SOLUTION 5
4. IMPLEMENTATION AND TESTING 7
5. SUMMARY 9
ACKNOWLEDGEMENTS 10
REFERENCES 11

LIST OF FIGURES

Figure **Page**

1. Forward code 4
2. Reverse code 6
3. Probability density function plot of First Passage Time 7
4. The first 100 samples generated in a sequence of 10,000 calls to FPT... 8
5. The last 100 samples generated by 10,000 calls to RFPT after the run shown in Fig. 4 8

ABSTRACT

We present a perfectly reversible method for *bi-directional* generation of samples from computationally complex probability distributions. While the previously best-known procedures consume memory proportional to the length of execution between changes of execution direction, here we present a scheme to completely eliminate the memory overhead. Our solution affords two important features, namely determinism and repeatability, across arbitrarily spaced changes of direction (and arbitrary number of samples) along the sample stream. We illustrate the perfect reversal method with *first passage time* distributions that appear in physical system models, and present its implementation and verification in FORTRAN.

1. INTRODUCTION

Here, we consider the challenge of designing a perfectly reversible solution for *bi-directional* generation of samples from computationally complex probability distributions. 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. While the best-known procedures consume memory proportional to the length of execution between changes of execution direction, here we present a scheme to completely eliminate 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: determinism and repeatability, across arbitrarily spaced changes of direction (and arbitrary lengths of sampling) along the sample stream.

Bi-directional execution of scientific codes can be used for efficient parallel execution on high performance computing platforms. 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. Here, we examine the memory cost in the bi-directional (reversible) execution of a commonly used probability distribution sampling method in scientific codes, namely, rejection sampling.

The rest of the document is organized as follows. In the rest of this section, the context for bi-directional execution and the use of probability distribution sampling are described, followed by the previously known use of reversible random number generators for reversal of simple probability distributions. This is followed by algorithmic detail of a commonly-used method called rejection sampling for sampling from more complex distributions. In Sect. 2, the implementation of the (traditional) forward execution is presented, illustrated in the context of sampling First Passing Time distributions. The perfect reversal of the forward execution is developed and presented in Sect. 3. An actual implementation and testing of the forward and reverse codes on the computer in the FORTRAN language is described in Sect. 4, finalized by a summary in Sect. 5.

1.1 BI-DIRECTIONAL EXECUTION

Bi-directional execution finds use in rollback-based high performance computing, debugging, and other areas [1-5]. However, bi-directional execution is very challenging to achieve. Naïve approaches can quickly render bi-directional execution impractical due to the tremendous memory costs of the execution traces needed to achieve reversibility [2, 6-11]. This problem is especially pronounced in high performance computing, due to the dramatic increases of trace lengths that result from a 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 probability distributions, and provide a new scheme that eliminates the memory cost for reversal.

1.2 SAMPLING PROBABILITY DISTRIBUTIONS

Probability distributions are 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 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, such as First Passage Time (FPT) 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.

1.3 REVERSIBLE RANDOM NUMBER GENERATORS

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 [2, 11-13]. 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, as we shall see, 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, the simple reversal of random number seed once does not work.

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 the 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.

1.4 REJECTION SAMPLING METHOD

We will now describe the rejection sampling method of generating random samples in greater detail. The rejection sampling method does not require the evaluation of the exact, closed-form (inverse of) the cumulative probability distribution function (CDF). Instead, it employs a sequence of progressively tighter upper $P_U^k(x)$ and lower $P_L^k(x)$ bounds, $k \geq 0, k \to \infty$, to the exact (inverse) probability distribution $P(x)$, such that $P_L^k(x) \leq P(x) \leq P_U^k(x)$. The bounds in the sequence must satisfy the condition of being progressively tighter: $P_L^k \leq P_L^{k+1}$, $P_U^k \geq P_U^{k+1}$, and $\left|P_U^{k+j} - P_L^{k+j}\right| < \left|P_U^k - P_L^k\right|$ for some $j > 0$. An algorithm that is based on these observations is shown next.

Rejection-based Sampling Algorithm:

1. Outer loop: Set $k = 0$. Generate a sample x belonging to P_U^0, and set $p = rP_U^0(x)$, where $0 < r \leq 1$ is a uniform random variate.
2. Inner loop: If $p > P_U^k(x)$ then reject the trial and go back to the first step (cycle Outer loop)
3. If $p \leq P_L^k(x)$, then accept the trial and return x (exit Outer loop)
4. Otherwise, increment k ($k \leftarrow k+1$) and go back to the second step (cycle Inner loop).

If the initial distribution is reasonably close to the actual distribution (i.e., $P_U^0 \approx P$) and the bounds are reasonably tight, then the rejection or acceptance will be achieved within a small number of iterations. The rejection sampling technique is typically enhanced with the objective of making the first few bounds as efficient to evaluate as possible.

2. FORWARD FORMULATION

In this section, we turn to a computer program subroutine for sampling FPT distributions [14] using the rejection sampling algorithm, and in Sect.3 develop a perfect reverse of the subroutine, followed by an implementation and verification in Sect. 4.

2.1 INVOCATION

The envisioned usage of the sampling subroutine is as follows: A scientific simulation program that simulates physical system models, at various moments during its execution, invokes the `FPT(t)` subroutine to obtain a sample from the first passage time distribution. The sample value is returned in the parameter `t`. After a sequence of (potentially scattered or intermittent) calls to `FPT()`, the program may desire to go back in simulation time (in speculative/optimistic synchronization mode [2, 5, 15, 16]), and hence would need to undo part of its sampling, resetting the position in the sample stream back by a few samples. Thus, a sequence of calls to the reverse subroutine `RFPT()` (designed later in the document) is invoked by the program. The objective is for the `RFPT()` calls to reverse the effects of `FPT()`and restore the state of the program to the same state that was present before the invocation of the corresponding `FPT()` calls.

We assume a reversible random number generator is used, by calling a subroutine called `RNG()`. Any suitable generator may be used, as long as a corresponding reversal subroutine called `RRNG()` is available to be used in the reverse formulation in Sect. 3.

2.2 SOURCE CODE

The source code for the FPT sampling subroutine is shown on the next page. It follows the algorithm outlined in Sect. 1.4, with the probability functions of the general algorithm customized for a First Passage Time distribution. An additional complication in this sampling code is the use of a different set of the lower and upper bound distributions. The choice of which set of functions to use for P_U^k and P_L^k is made at runtime, based on a threshold on the generated uniform random value, qualitatively corresponding to “long time” and “short time” spans of first passage. Using a conditional statement, the choice is made for every candidate sample at run time to decide which functions are appropriate for that random value.

```
! Compute First Passage Time and return it in t
subroutine FPT(t)
   real(kind=r_wp), intent(out) :: t

   real(kind=r_wp) :: pi,tau,Fs0,Fl0,cut
   parameter(pi = 3.14159265358979323846d0)
   parameter(tau = 0.0796d0)
!     Fs0 = 2*erfc(1/sqrt(16*tau))
!     Fl0 = 4/pi*exp(-pi^2*tau)
   parameter(Fs0 = 0.42031187432794d0)
   parameter(Fl0 = 0.58038939207732d0)
   parameter(cut = Fs0/(Fs0+Fl0))
   real(kind=r_wp) :: r,y0,y,fi,dfi,beta
   integer k

   OuterLoop: do
      call RNG(r)
      if(r .lt. cut) then
!     Short time pick
         r = r/cut
         t = 1d0/(16 * dierfc(0.5d0*r*Fs0)**2)
         y0 = 1d0/sqrt(4*pi*t**3) * exp(-0.0625d0/t)
         call RNG(r)
         y = r*y0
         fi = 0.5d0 * exp(-0.0625d0/t) / sqrt(pi*t**3)
         k = 1
         InnerLoop_ShortPick: do
            k = k+1
            dfi = (k+0.5d0)*exp(-0.25d0*(k+0.5d0)**2/t)/sqrt(pi*t**3)
            if(y .gt. fi) cycle OuterLoop !Rejection
            fi = fi - dfi
            if(y .le. fi) return !Acceptance

            k = k+1
            dfi = (k+0.5d0)*exp(-0.25d0*(k+0.5d0)**2/t)/sqrt(pi*t**3)
            if(y .le. fi) return !Acceptance
            fi = fi + dfi
            if(y .gt. fi) cycle OuterLoop !Rejection
         enddo
      else
!     Long time pick
         r = (r-cut)/(1d0-cut)
         t = tau - log(r)/pi**2
         beta = pi**2 * t
         y0 = 4*pi*exp(-beta)
         call RNG(r)
         y = r*y0
         fi = 4*pi*exp(-beta)
         k = 1
         InnerLoop_LongPick: do
            k = k+2
            dfi = 4*pi*exp(-beta * k**2)
            if(y .gt. fi) cycle OuterLoop !Rejection
            fi = fi - dfi
            if(y .le. fi) return !Acceptance

            k = k+2
            dfi = 4*pi*exp(-beta * k**2)
            if(y .le. fi) return !Acceptance
            fi = fi + dfi
            if(y .gt. fi) cycle OuterLoop !Rejection
         enddo
      endif
   enddo OuterLoop
end subroutineReversal
```

Fig. 1. Forward code.

3. REVERSAL

3.1 OBSERVATIONS

Observation 1: The only side-effects of the FPT( ) routine is the modification of the random number seed. All other variables are temporary storage, and consequently, no reversal needs to be applied to them. In other words, temporary values initially contain *don't-care* values, which do not need to be restored exactly.

Observation 2: Every call to FPT( ) results in invocation of RNG( ) exactly two times per iteration of the outer loop. As a regular expression, the following invariant is preserved by the loop: (R1, [R2a,R2b])+, where [*x*,*y*] denotes either *x* or *y*, R1 is RNG invoked at line 17, R2a is RNG at line 23 and R2b is RNG at line 46.

Given that we know how to reverse an individual RNG call, the reversal problem of FPT( ) becomes that of detecting how many invocations of RNG to be reversed for a given call to FPT( ). The underlying problem with perfect reversal is that the loop in the forward computation could execute any number of iterations, greater than or equal to unity. If we "remember" exactly how many iterations of the loop have been made in a given call to FPT( ), then it is straightforward to reverse FPT( ). The reversal is simply realized by invoking the reverse version of RNG twice the number of iterations. The drawback of this approach is that an integer variable needs to be allocated for each call to FPT( ), in order to remember the iteration count for that call. This makes the storage requirements for reversal to become proportional to the number of FPT( ) calls, which can get quite large in a simulation that makes very many calls to FPT( ). For example, with millions of particles and many millions of passage time computations, the storage requirements can become quite large. Ideally, we are interested in a reversal that uses absolutely no storage. Such a reversal is called perfect reversal [11]. The question becomes: Is FPT( ) computation perfectly reversible? Our answer to this is in fact in the positive. We will now show how to perfectly reverse FPT( ) calls.

3.2 SOLUTION

We will first consider the easy part of the solution. Any single iteration of the outer loop can be perfectly reversed simply by invoking reverse RNG( ) twice. This follows from observations 1 and 2. Observation 1 helps us focus only on RNG seeds for prefect reversal. Observation 2 helps us realize that there are exactly two RNG( ) calls per iteration. Thus, it is sufficient to invoke reverse RNG( ) two times to restore the state perfectly across iterations. The more difficult question now becomes the problem of discovering how many iterations were executed in the FPT( ) call? If we knew this count, we simply invoke the RNG( ) two times per iteration, for that many iterations. That would restore the state perfectly.

The complicating factor for the loop is that, when we reverse one iteration of the loop (by invoking inverse RNG two times), we are still not sure whether we reached the beginning of the loop of the forward execution, or if we need to continue to reverse additional iterations of the loop. The iteration count is not obvious. The detection of reaching the beginning iteration of the forward execution of the loop becomes the main problem. Fortunately, one additional insight helps relieve this dilemma: any iteration in which the "return" call is invoked (at lines 32, 36, 55 or 59) is clearly the last iteration of that FPT( ) call.

Observation 3: Moreover, this termination condition depends only the state of that iteration alone, and does not depend on state from prior iterations. We capitalize on this termination condition, by temporarily reversing one iteration backward and seeing if the test succeeds on that previous state. If the condition succeeds, then it implies that that iteration belonged to a previous FPT( ) call, and not to the current call. On the other hand, if the test is negative (i.e., it shows that the loop does not terminate at that iteration) it shows that the iteration belongs to the current FPT( ) call which is being reversed. This termination test essentially helps us determine if we need to jump back one more iteration or if we are done reversing the FPT( ) call that is being reversed. The source code for the reversal is shown on the next page.

```fortran
! Reverse the effects of a call to FPT
subroutine RFPT()
   real(kind=r_wp) :: t, pi,tau,Fs0,Fl0,cut
   parameter(pi = 3.14159265358979323846d0)
   parameter(tau = 0.0796d0)
   parameter(Fs0 = 0.42031187432794d0)
   parameter(Fl0 = 0.58038939207732d0)
   parameter(cut = Fs0/(Fs0+Fl0))
   real(kind=r_wp) :: r,y0,y,fi,dfi,beta
   integer k
   OuterLoop: DO
!Reverse the most recent iteration; guaranteed to exist, since niterations>=1
      call RRNG(r)
      call RRNG(r)
!Now, see if we need to undo more iterations
!Tentatively undo one iteration backward
      call RRNG(r)
      call RRNG(r)
!Verify if this reversed iteration belongs to this call, or is last of
previous
!This is done by logically (temporarily) re-computing that iteration
      call RNG(r)
      if(r .lt. cut) then
         r = r/cut
         t = 1d0/(16 * dierfc(0.5d0*r*Fs0)**2)
         y0 = 1d0/sqrt(4*pi*t**3) * exp(-0.0625d0/t)
         call RNG(r)
         y = r*y0
         fi = 0.5d0 * exp(-0.0625d0/t) / sqrt(pi*t**3)
         k = 1
         InnerLoop_ShortPick: do
            k = k+1
            dfi = (k+0.5d0)*exp(-0.25d0*(k+0.5d0)**2/t)/sqrt(pi*t**3)
            if(y .gt. fi) cycle OuterLoop
            fi = fi - dfi
            if(y .le. fi) exit OuterLoop
            endif
            k = k+1
            dfi = (k+0.5d0)*exp(-0.25d0*(k+0.5d0)**2/t)/sqrt(pi*t**3)
            if(y .le. fi) exit OuterLoop
            fi = fi + dfi
            if(y .gt. fi) cycle OuterLoop
         enddo
      else
         r = (r-cut)/(1d0-cut)
         t = tau - log(r)/pi**2
         beta = pi**2 * t
         y0 = 4*pi*exp(-beta)
         call RNG(r)
         y = r*y0
         fi = 4*pi*exp(-beta)
         k = 1
         InnerLoop_LongPick: do
            k = k+2
            dfi = 4*pi*exp(-beta * k**2)
            if(y .gt. fi) cycle OuterLoop
            fi = fi - dfi
            if(y .le. fi) exit OuterLoop
            k = k+2
            dfi = 4*pi*exp(-beta * k**2)
            if(y .le. fi) exit OuterLoop
            fi = fi + dfi
            if(y .gt. fi) cycle OuterLoop
         enddo
      endif
   enddo OuterLoop
end subroutineImplementation and Testing
```

Fig. 2. Reverse code.

4. IMPLEMENTATION AND TESTING

The `FPT()` and `RFPT()` subroutines were exercised in a test program in which several calls to `FPT()` are invoked after which `RFPT()` is invoked as many times, and the results compared. The initial sample generated by `FPT()` is verified to match the final sample generated by `RFPT()`. In a sequence of over 10 million invocations, no precision problems are observed from round off errors or other numerical problems. This is to be expected, since (a) the periods of the random number generator are very long, and (b) the time computation possesses a deterministic, one-to-one mapping from a random number seed to the computed floating point value for time.

Also, no additional memory is allocated by the subroutines during the invocations, thus demonstrating the routines' independence from invocation length.

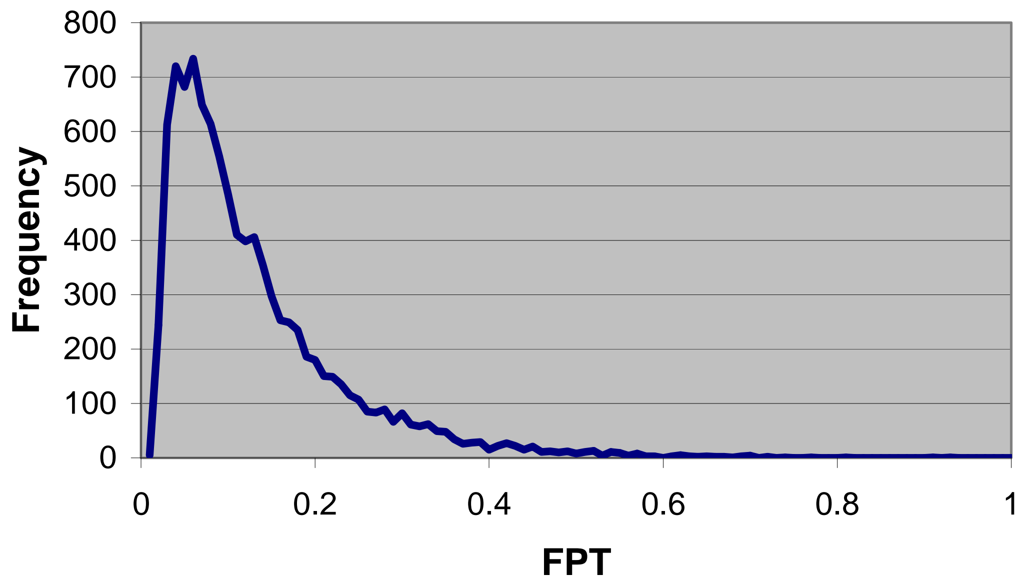

Fig. 3. Probability density function plot of First Passage Time.

Fig. 3 shows the probability density function of FPT plotted for 10,000 throws, binned in equal, regularly-spaced intervals of size *dt*=0.01.

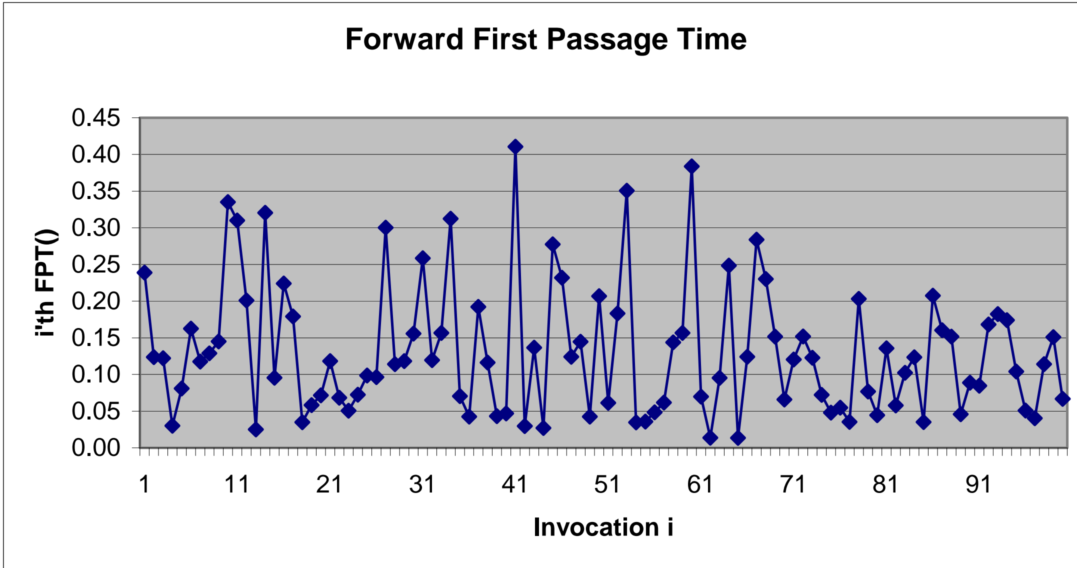

Fig. 4. The first 100 samples generated in a sequence of 10,000 calls to FPT.

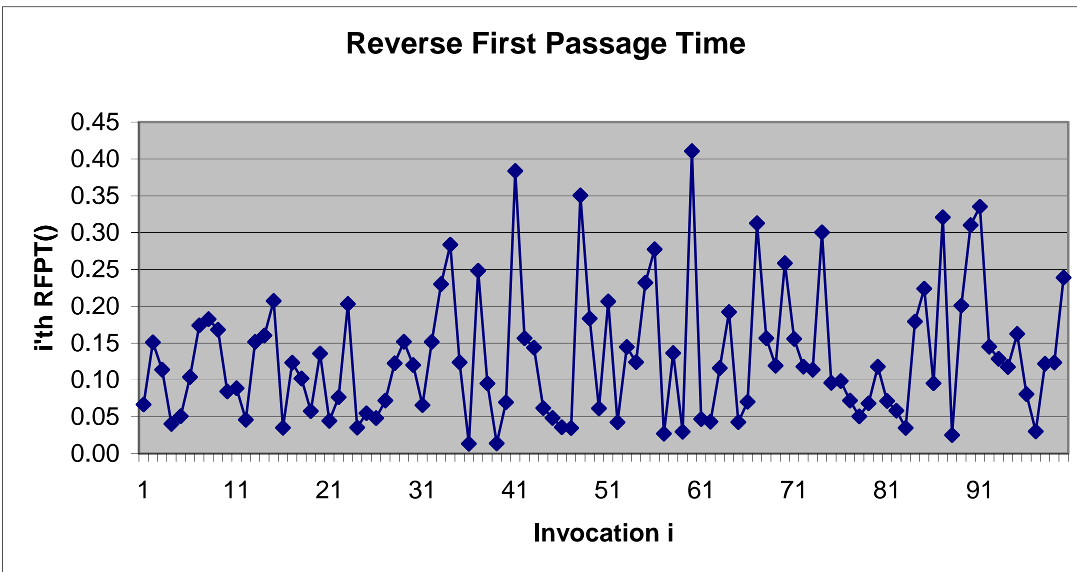

Fig. 5. The last 100 samples generated by 10,000 calls to RFPT after the run shown in Fig. 4.

As an illustration of the effects of forward and reverse execution, Fig. 4 shows the first 100 samples in a sequence of 10,000 samples generated by `FPT()` in the forward direction. At the end of the generation of the 10,000 samples, the sequence is reversed 10,000 times by invocation of `RFPT()`. Fig. 5 shows the final 100 values generated by `RFPT()`. As expected, the samples are regenerated backwards exactly in the reverse direction, and thus, the samples in Fig. 5 form a mirror image of those in Fig. 4.

5. SUMMARY

In physical system models that rely on certain distribution sampling methods, reversibility needs to be enabled, ideally with little or no memory cost. However, reversibility properties of complex sampling codes have been largely unexplored, making checkpointing-based methods the only reversal alternative. Here, we examined the reversibility aspects of a class of distribution sampling routines known as rejection sampling. We showed that, despite the apparent complexity of the code, such codes can be perfectly inverted. We illustrated the reversal with an instance of the rejection sampling method that is used to sample the FPT distribution. The net effect of our findings is that memory trace is completely eliminated in enabling perfectly reversible sampling, enabling forward or backward movement in the sampled stream with constant memory cost.

ACKNOWLEDGEMENTS

This paper has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the U.S. Department of Energy. Accordingly, 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, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes.

This effort has been supported by research sponsored by the Laboratory Directed Research and Development Program of Oak Ridge National Laboratory, managed by UT-Battelle, LLC, for the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

REFERENCES

1. P. Bishop, *Using Reversible Computing to Achieve Fail-Safety*. in *ISSRE-97*. 1997. IEEE Computer Society Press.
2. 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.
3. B. Boothe, *Efficient Algorithms for Bidirectional Debugging*. in *Programming Language Design and Implementation*. 2000. ACM Press.
4. J. Lee, et al., *Reversible Computation in Asynchronous Cellular Automata.* Lecture Notes in Computer Science, 2002. **2509**: p. 220–229.
5. 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.
6. 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.
7. 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.
8. 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.
9. D. West and K. Panesar, *Automatic Incremental State Saving*, in *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*. 1996. p. 78–85.
10. F. Gomes, *Compiler Techniques for State Saving in Parallel Discrete Event Simulation*, in *Computer Science*. 1996, University of Calgary, Canada.
11. K. S. Perumalla, *Techniques for Efficient Parallel Simulation and their Application to Large-scale Telecommunication Network Models*, in *College of Computing*. 1999, Georgia Institute of Technology: Atlanta. p. 150.
12. K. Perumalla, R. Fujimoto, and A. Ogielski, *TeD - A Language for Modeling Telecommunications Networks.* Performance Evaluation Review, 1998. **25**(4).
13. K. S. Perumalla and R. M. Fujimoto, *Source Code Transformations for Efficient Reversibility*. 1999, College of Computing, Georgia Institute of Technology: Atlanta.
14. S. Redner, *A Guide to First-Passage Processes*. 2001: Cambridge University Press
15. R. M. Fujimoto, *Optimistic Approaches to Parallel Discrete Event Simulation.* Transactions of the Society for Computer Simulation, 1990. **7**(2): p. 153–191.
16. K. S. Perumalla, *μsik - A Micro-Kernel for Parallel/Distributed Simulation Systems*. in *Workshop on Principles of Advanced and Distributed Simulation*. 2005.