NON-UNIFORM RANDOM NUMBER GENERATION THROUGH PIECEWISE LINEAR APPROXIMATIONS

David B. Thomas and Wayne Luk

Department of Computing,
Imperial College, London
email: {dt10,wl}@doc.ic.ac.uk

ABSTRACT

This paper presents a hardware architecture for non-uniform random number generation, which allows the generator's distribution to be modified at run-time without reconfiguration. The architecture is based on a piecewise linear approximation, using just one table lookup, one comparison and one subtract operation to map from a uniform source to an arbitrary non-uniform distribution, resulting in very low area utilisation and high speeds. Customisation of the distribution is fully automatic, requiring less than a second of CPU time to approximate a new distribution, and around 1000 cycles to switch distributions at run-time. Comparison with Gaussian specific generators show that the new architecture uses less than half the resources, provides a higher sample rate, and retains statistical quality for up to 50 billion samples, but can also generate other distributions.

1. INTRODUCTION

Many computationally intensive applications have no closed form solution, and rely on Monte-Carlo simulations or stoch–astic algorithms to provide approximate solutions. As the problems to be solved become larger and more sophisticated, it is becoming infeasible to execute the applications on a conventional CPU cluster due to the high cost and power consumption. One direction that is now being explored is to move such applications into reconfigurable hardware accelerators, which have many possible advantages in terms of cost, power, physical space, and execution time [1, 2].

A key component of any hardware Monte-Carlo simulation is a high-performance non-uniform Random Number Generator (RNG), which provides a high sample rate while using the minimum possible hardware resources. Current approaches to non-uniform RNGs in hardware have focused on analytically defined distributions, such as the Gaussian distribution, where it is possible to directly customise the hardware architecture to implement an analytically derived transform. These fixed generation architectures provide high performance, but only produce a single distribution, and require reconfiguration in order to switch to a different distribution.

Stock	Mean	Std-Dev	Skewness	Kurtosis
GE	0.0008	0.028	-0.16	0.6
XLNX	0.0009	0.043	-0.09	1.5
ALTR	0.0013	0.045	-0.19	2.5
MSFT	0.0018	0.031	-0.59	10.3

Table 1. Statistical properties of the daily log-returns for four different equities over the same time period.

In some applications, such as Bit Error Rate (BER) testing, only one distribution is needed, but many other applications require more than one distribution, including empirically derived distributions. For example, in financial computing historical data is used to provide an empirical distribution for future events. Table 1 gives statistics for the log-returns of four different equities over the same time period. A common approximation is to assume that the log-returns of equities are normally distributed, with both kurtosis (asymmetry) and kurtosis (pointiness) equal to zero, but this is clearly not the case: the skewness and kurtosis of the Microsoft log-returns shows that a normal approximation is not appropriate. To provide an accurate simulation of a portfolio involving these four stocks it is necessary to be able to generate approximations to all four distributions, as well as to adjust those distributions as new data becomes available.

This paper presents a new hardware architecture for non-uniform random number generation that supports rapid switching between probability distributions at run-time. The key benefits of this approach are:

- a fast and area efficient hardware architecture using only memory look-ups, comparisons and additions;
- the ability to quickly change the generated distribution at run-time without using reconfiguration;
- a fully automatic method for approximating new distributions in near real-time.

1-4244-0 312-X/06/$20.00

2. BACKGROUND

The probability distribution of a continuous random variable X can be described using it's Cumulative Distribution Function (CDF) $F(x)$, which provides the probability that X is less than some value x and monotonically increases from $F(-\infty) = 0$ to $F(+\infty) = 1$. The CDF is the integral of the distribution's Probability Density Function $f(x)$, which measures the point-wise probability of each value:

$$F(x) = \Pr[x < X] = \int_{-\infty}^{x} f(x) \tag{1}$$

Non-uniform RNGs often use two steps: uniform random number generation, which provides the underlying randomness, followed by a transformation which converts the uniform distribution to the non-uniform target distribution. The most direct transform is the inversion method, which assumes the existence of an Inverse Cumulative Distribution Function (ICDF). This uses uniformly distributed random numbers $x_1, x_2, ...$ in the range $[0, 1]$, then applies the ICDF to give $y_i = F^{-1}(x_i)$. As there is no closed-form solution for the ICDF of many useful distributions, F^{-1} is usually a numerical approximation to the true ICDF.

In software the inverse method is used to create general purpose (i.e. distribution independent) random number generators, for example by using piecewise Hermite interpolation of the ICDF [3], or by creating ratio-of-uniform based generators [4], but these methods use high-precision floating-point maths, and so are not appropriate for hardware implementation. Inversion methods in hardware have concentrated on table based approximations to the ICDF using low-order piecewise polynomial approximation, with either a regular domain partitioning scheme and large tables [5], or irregular domain partitioning to allow smaller tables [6]. However, these methods provide poor PDF accuracy given the large amounts of memory and resources needed.

Most work on hardware RNGs has focused on distribution specific techniques, particularly for the Gaussian distribution. The most commonly used is the Box-Muller method, which transforms two independent uniform random variables into two independent Gaussian variables using a pair of function transforms, requiring the evaluation of $\ln$, $\sqrt{}$ and $\sin$. The transform can be implemented in hardware using two separate function approximation steps, with the most sophisticated approach using irregular domain partitioning and error analysis to guarantee PDF accuracy for over 10^{10} samples [7]. The Ziggurat method is another Gaussian specific transform, based on the rejection method. This means that a set of candidate samples are generated, but then a fraction of these samples are then discarded. It uses a set of table look-ups to reduce the average work per-sample as much as possible, and has also proven to be efficient in hardware as well as software [8]. The Wallace generator uses a novel approach to produce Gaussian samples directly, rather than transforming uniform samples. It has an efficient hardware implementation [9], but also has some statistical flaws.

To provide the source randomness for the transform step a uniform RNG is also needed; this must have both a long period and good statistical quality, so that the non-uniform distribution will not be biased. The most common hardware technique is the Linear Feedback Shift Register (LFSR) [10], which uses a binary linear recurrence based on a primitive polynomial to provide a period of $2^n - 1$ from an n-bit state. However, LFSRs provide poor area utilisation, as only one bit can be used from each LFSR, and the statistical quality of the generated sequence is poor. Non-linear recurrences based on Cellular Automata (CA) have also been used in hardware [11], and provide better area utilisation than LFSRs as more than one bit can be taken from each generator. However, the theoretical properties of CAs are not well understood, and it is difficult to guarantee properties such as generator period and quality.

Binary linear recurrences using more sophisticated feedback schemes than the LFSR provide better area utilisation and statistical quality. The Combined Tausworthe combines the output of multiple LFSRs to provide multiple bits at each iteration, and though intended for word-based operations is also efficient in hardware [7, 8]. A more efficient (though more complex to implement) linear recurrence is also possible, which is optimised for LUT based architectures [12]. These LUT-optimised linear recurrences provide n output bits per cycle, using a total of n LUTs and n FFs, and have a period of $2^n - 1$, with good statistical quality.

3. APPROXIMATION ALGORITHM

The non-uniform generation technique presented in this paper does not use the inversion or rejection techniques, but instead uses a mixture of simple distributions to approximate a more complex target distribution. In this section the generation algorithm is presented, along with the algorithm used to initialise generators with a target distribution.

The proposed generation algorithm uses the fact that complex distributions can be formed from mixtures of multiple distributions: given a target CDF and PDF $T(x)$ and $t(x)$, we can calculate a set of n weights $w_1..w_n \in [0, 1]$ and a set of component PDFs $g_1..g_n$, such that the weighted combination $m(x)$ of the component PDFs equals the target PDF:

$$t(x) \equiv m(x) = \sum_{i=1}^{n} w_i g_i(x) \tag{2}$$

This allows samples from $t(x)$ to be generated, by randomly selecting one of the components according to their weights, then sampling the selected component distribution.

It is always possible to exactly approximate $t(x)$ in this way, but for most distributions of interest this requires that at least one of the component distributions is as complex as the

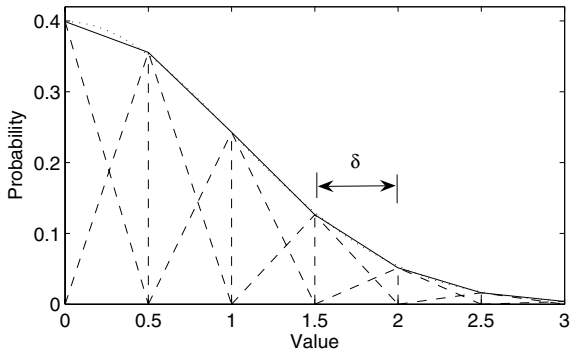

Fig. 1. A simple approximation to the Gaussian distribution, using triangles with $\delta = 0.5$.

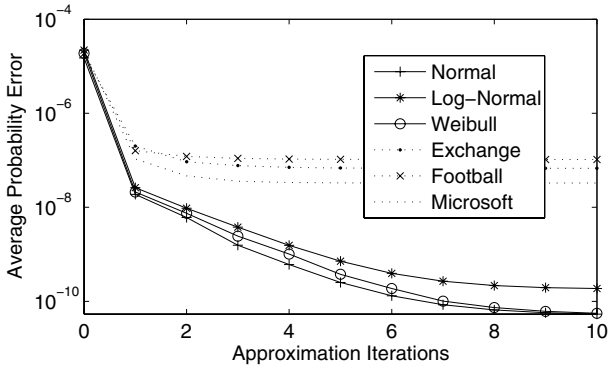

Fig. 2. Average per-value approximation error (over 2^{16} output values), for approximations to analytic and empirically derived distributions.

target PDF to calculate. The approach used here is not to approximate the target distribution exactly, but instead to use a large number of very simple component distributions that provide an acceptably good approximation to the target distribution. In [13] this approach was used with homogeneous weights and heterogeneous components: each component had an equal probability of being selected, but components of different distributions (triangle, rectangle and trapezoid) could be used, with arbitrary per-component variances and means. The drawback of this approach is that selecting a good set of component distributions requires a complex optimisation strategy, so it takes a significant amount of time (from minutes to hours) to approximate each distribution.

In this paper the opposite approach is taken: a homogeneous set of components is used, but the weights (selection probability) assigned to each component can be varied. This provides a much smaller design space to be explored for each target distribution, allowing a simpler and faster algorithm to be used to generate approximations for each target distribution. The component distributions used are all triangle distributions, as the triangular distribution can be easily and exactly generated by adding two uniform samples together. All the component triangles have the same width δ, and are spaced at increments of δ, i.e. the mean of component i is $i\delta + k$ (where k is a scaling offset), so the PDF of the mixture is simply the linear interpolation of the triangle weights, and can be calculated exactly. Figure 1 shows a simple example where 7 components are used to approximate the Gaussian distribution with $\delta = 0.5$.

A first approximation to the weights $w_1..w_n$ can be provided by evaluating the target PDF at each triangle's center, $w_i = t(i\delta + k)$, followed by a normalisation to make sure that the weights add up to one. However, this does not provide the best possible fit, as it ignores the PDF error at points that do not lie on the triangle mid-points. To achieve a better fit it is necessary to slightly increase the approximation error at the mid-points, providing a much better overall fit. One method for calculating the best approximation is to use solve a system of linear equations, as used in a similar approach to approximating the Gaussian distribution [14], but this was found to be inaccurate when applied to highly skewed or leptokurtic distributions, and is computationally infeasible when large numbers of triangles are used.

Our approach is to optimise the approximation error by iteratively smoothing the errors out, by finding points with large errors, then spreading the error across the rest of the distribution. We assume that the target distribution is relatively smooth, at least on the scale of δ, so it is sufficient to only measure approximation with a granularity of $\delta/2$. The iterative algorithm used is:

1. Set $w_1...w_n$ to the first approximation using $t(x)$.
2. Set $c = i\delta/2 + k$, for $1 \leq i \leq 2n + 1$.
3. Calculate $p = t(c)$, the vector of target probabilities.
4. Calculate $d = p - m(x)$, the difference between the current approximation and the target.
5. Find the largest magnitude error e_i within d.
6. Subtract the error e_i from the contributing triangle weight(s): $w_{i/2}$, if e_i is on a triangle mid-point, or $w_{\lfloor i/2 \rfloor}$ and $w_{\lfloor i/2 \rfloor + 1}$ if e_i is between two triangles.
7. Update d to reflect the new triangle weightings, and calculate $d = d/\sum(d)$ to normalise the differences.
8. Use the normalised differences to mix the excess error back in: $w_k = e_i(1/2d_{2k-1} + d_{2k} + 1/2d_{2k+1})$.
9. If the solution is not acceptable return to Step 4.

The algorithm is terminated either when no more improvement is made, or after a maximum number of iterations.

Figure 2 shows the approximation error for six distributions against the number of optimisation iterations that are performed. The distribution range is quantised into 2^{16} discrete values, and the error is measured as the average absolute probability error across all the values. The *Exchange*, *Football*, and *Microsoft* are all empirically defined, and the

optimiser cannot achieve the lower overall error achieved with the smoother analytically defined distributions.

When sampling from the mixture it is necessary to randomly select one of the component distributions according to the weights. This requires a discrete random variate I, with Probability Mass Function (PMF) $\Pr[i = I] = w_i$. Efficient sampling from a discrete distribution with arbitrary probabilities can be achieved in $O(1)$ time and $O(n)$ storage using Walker's alias method [15].

The alias method makes use of two tables, calculated in $O(n)$ time, one of which contains probability thresholds between zero and one ($t_1..t_n$), while the other contains alternate indices from 1 to n ($a_1..a_n$). To sample from the distribution a uniform random integer $i \in [1, n]$ is first generated, which selects a threshold t_i and alternate index a_i. A continuous uniform random variable x in the range $[0, 1]$ is then also generated. If $x < a_i$, then i is used as the selected sample from I,7 otherwise the alternate index a_i is used.

The calculation of target PDF, optimisation of triangle weights, and alias table calculation all scale linearly with the number of triangles. The entire distribution approximation step takes less than a second in software for all $n \leq 2^{14}$, and significantly less for $n \leq 2^{12}$, which is the range used within this paper. This fast approximation allows distributions to be updated at run-time, for example financial simulations can approximate the distribution of stocks before each execution to take into account the most recent intra-daily figures.

The overall approximation algorithm can now be described in full. The weights $w_1..w_n$ are only needed during the setup stage, and are not used when generating samples. This leaves three scalar constants and two vectors that are needed at run-time: n, the number of triangles; δ, the distance between adjacent triangles; k, a real correction factor used to adjust the distribution mean; $t_1..t_n$, probability thresholds between 0 and 1; and $a_1..a_n$ integer alternate indices between 1 and n. The generation algorithm is:

1. Generate i, a uniform random integer between 1 and n, and y, a uniform random real between 0 and 1.
2. If $y > t_i$ then set $i \leftarrow a_i$.
3. Calculate $c \leftarrow i\delta + k$, center of the selected triangle.
4. Generate uniform random reals $z_1, z_2 \in [0, \delta]$.
5. Return $c + z_1 - z_2$, a random sample from $m(x)$.

4. HARDWARE IMPLEMENTATION

The generation algorithm presented in the previous section has three advantages that make it ideal for hardware implementation. First, there are no feedbacks or data-flow hazards, allowing the generator to be completely pipelined. Second, only one memory access is needed per generated sample (assuming a_i and t_i are packed together). Finally, the only operations required are comparison and subtraction.

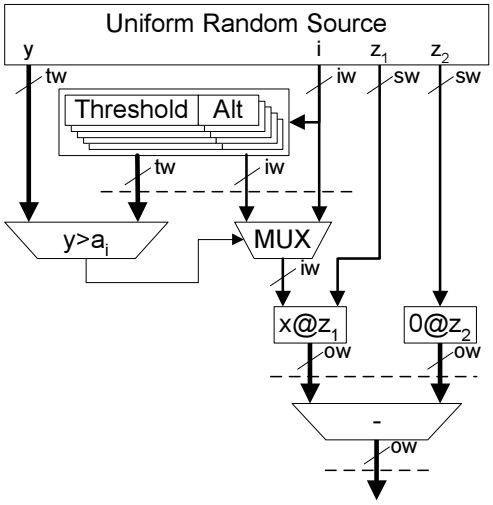

Fig. 3. Hardware architecture of the generator algorithm.

The last point is not true in general, due to the multiplication by δ in Step 3, but in the specific architecture presented here the multiplication is replaced with shifts by requiring that $\delta = 2^{-x}$ for some integer x.

The first step in translating the algorithm to hardware is to assign widths to the variables. The index variables ($a_1..a_n$ and i) must have width $iw = \lfloor log_2 n \rfloor$, as they address triangles $1..n$. The threshold variables ($t_1..t_n$ and y) are quantised to an arbitrary fractional width tw; the minimum triangle height is limited to 2^{-tw}, so this choice affects the ability of the generator to approximate the tails of distributions. The random numbers used to create the triangle distribution, z_1 and z_2, need to be generated with $sw = log_2(\delta)$ bits (using the fact that δ is restricted to a binary power). The overall output of the generator is then $ow = iw + sw$ bits wide, which can be interpreted as either signed or unsigned.

In the architecture presented here the offset k is not used, as the n triangles completely cover the generator's output range. This has the advantage of simplicity and area efficiency, but might not be appropriate when approximating distributions that have significantly different means and small variances, as many of the triangles (i.e. table entries) would be wasted on zero probability parts of the range, leaving only a small number of triangles over the non-zero probability areas. In such cases it would make sense to have $ow > iw + sw$, and to use an ow width k to move the approximated range around within the total output range.

Figure 3 shows the architecture of the generator, including the widths of the signals (the @ operator represents bitwise concatenation). Note that the separate additions and subtractions from Step 5 of the algorithm have been combined into a single subtraction. The required resources of the generator are thus: one tw bit comparator, one iw multiplexer, one ow bit subtracter,a RAM containing n elements

n	64		512		2048		1024
ow	12		16		24		16
tw	8	16	16	27	16	25	26
RAMs	-	-	0.5 (1)		3	4	1 (2)
LUTs	94	114	83	105	113	131	182
FFs	52	61	61	72	87	96	211
Slices	63	69	51	62	69	80	137
MHz	250	219	182	183	172	168	249
Norm	16	18	31	34	35	+	36
LgNrm	12	14	27	31	32	+	35
Weib	16	17	28	31	34	+	36
Foot	12	13	23	24	30	33	31
Exchg	11	11	25	27	29	32	30
Msft	14	15	24	26	34	35	33

Table 2. Area and resource utilisation of generators in the Virtex-II architecture, and the number of samples (log_2) before failure of the χ^2 test against different distributions.

of width $tw + iw$, and a random number generator capable of supplying $rw = tw + iw + 2sw$ independent random bits per cycle. Using the LUT-optimised linear recurrence technique [12] an rw bit generator requires exactly rw LUTs and rw FFs. In total the LUT usage can be estimated as $4iw + 3tw + 3sw$ LUTs.

The critical path will be from the random index i, through the RAM, into comparator, through the MUX, then down through to the output. However, the generator can be heavily pipelined, and suggested pipeline registers that use already available FFs are shown in Figure 3 as heavy dashed lines. Note that it is not necessary to synchronise the random bits when pipelining, as long as the bits have not already been used in some previous stage.

A $rw = tw + iw + 2sw$ bit wide uniform bit generator, will have a generator period of $2^{rw} - 1$. However, if rw is small (e.g less than 64), this will result in a low period generator with poor statistical quality. One solution is to use a random bit generator with more bits than are needed, then to ignore the excess bits. A better solution is to share a random bit generator between two or more non-uniform generators. For example, if two generators each need $rw = 40$ bits, then a single 80 bit random bit generator can be shared, keeping the cost per non-uniform generator at rw bits, but providing a higher quality random bit source to both.

5. EVALUATION

Table 2 shows the performance of a selection of generators in the Virtex-II architecture, generated through a single parameterised Handel-C core. The code was written with portability and area efficiency in mind, and so does not achieve the best possible performance. The final generator (on the right of the table) was manually optimised for performance, and achieves a much higher clock rate, at the expense of more than doubling the required area. Where block-RAM ports can be shared with another generator the proportion used by one generator is shown, with the total amount needed shown in brackets.

Underneath the performance figures, an evaluation of the goodness-of-fit of the different generators is presented, using a χ^2 test. The test is performed for $s = 2^4, 2^5, 2^6, ..., 2^{36}$ samples, using $\sqrt{s}$ equal probability buckets for each set of samples (adjusted for the discrete range), until the test fails at the 5% level. The table shows $log_2(s)$, the point at which the test was failed, or + if the test did not fail with 2^{36} samples. All the generators perform better when approximating analytic distributions rather than empirical distributions, due to their smoothness. The empirical distribution's CDFs were only lightly smoothed, and so contained many lumps and bumps in the PDF that are difficult to approximate: in practice it is likely that much more smoothing would be applied to empirical distributions, which would allow better approximations to be made.

Table 3 provides a comparison between the performance of the speed-optimised piecewise generator from Table 2 and other types of Gaussian random number generator, all implemented in the Virtex-II architecture. Where resources can be shared, or more than one sample is generated per cycle, the figures are normalised for a single cycle per sample. χ^2 failure points marked with a "+" indicate the largest sample size tested, so the actual failure point may be much higher. Although the piecewise generator cannot produce the same quality level as the dedicated Gaussian generators, it still produces up to 2^{36} samples before failing the χ^2 test, as well as providing the highest sample rate and using less than half the resources needed by the other generators. The piecewise generator is also able to switch to other distributions in 1024 clock cycles: all the other generators are limited to just the standard Gaussian distribution, and require extra logic even to change the distribution variance.

6. CONCLUSION

This paper has presented a non-uniform random number generator architecture which approximates probability distributions using discrete mixtures of triangular distributions. The key advantages of this technique are:

- Low resource usage: only table lookups, comparison, and subtraction are needed, so only standard logic resources plus a small amount of RAM is needed.
- High speed: the architecture contains no feedback and can be heavily pipelined.
- Distribution modification at run-time: the generator distribution can be modified at run-time by modifying RAM contents, so switching distributions occurs without reconfiguration.

Class	Previous Hardware Methods				Piecewise	Software	
Method	Box-Muller	Wallace	Ziggurat	Trapezoid	$n = 2^{10}$,$tw = 26$	Ziggurat	Wallace
Reference	[7]	[9]	[8]	[13]	-	[16]	[17]
Output Width	16	24	32	17	16	-	-
Slices	757	770	891	451	137	-	-
RAMs	1.5	6	4	3	1	-	-
DSPs	6	4	2	-	-	-	-
Rate (MS/s)	202	155	168	194	249	37	81
χ^2 Fail (Samples)	2^{40}+	2^{36}+	2^{30}+	2^{30}	2^{36}	-	-

Table 3. Comparison of different methods for generating Gaussian random numbers.

- Fast approximation of arbitrary distributions: both analytically and empirically defined distributions can be approximated in software in less than a second using the described optimisation process.

When compared with existing Gaussian generator architectures, the piecewise technique uses less than half the logic resources, achieves a 25% faster generation rate, and is able to switch to other distributions at run-time. While the accuracy of the produced distribution is not as high as the dedicated techniques the quality is sufficient for tens of billions of samples. As a result the technique is ideal for applications where many different distributions must be generated within a single application, but the total number of samples consumed from each distribution is not large, as for example in certain types of financial simulation.

7. REFERENCES

[1] G. L. Zhang, P. H. W. Leong, C. H. Ho, K. H. Tsoi, D.-U. Lee, R. C. C. Cheung, and W. Luk, "Reconfigurable acceleration for monte carlo based financial simulation," in *International Conference on Field-Programmable Technology*. IEEE Computer Society Press, 2005, pp. 215–224.

[2] A. Negoi and J. Zimmermann, "Monte Carlo hardware simulator for electron dynamics in semiconductors," in *International Annual Semiconductor Conference*, Sinaia, Romania, 1996, pp. 557–560.

[3] W. Hörmann and J. Leydold, "Continuous random variate generation by fast numerical inversion," *ACM Transactions on Modeling and Computer Simulation*, vol. 13, no. 4, pp. 347–362, 2003.

[4] J. Leydold, "Short universal generators via generalized ratio-of-uniforms method," *Mathematics of Computation*, vol. 72, no. 243, pp. 1453–1471, 2003.

[5] J. M. McCollum, J. M. Lancaster, D. W. Bouldin, and G. D. Peterson, "Hardware acceleration of pseudo-random number generation for simulation applications," in *IEEE Southeastern Symposium on System Theory*, 2003, pp. 299–303.

[6] E. Boutillon, J.-L. Danger, and A. Ghazel, "Design of high speed AWGN communication channel emulator," *Analog Integrated Circuits and Signal Processing*, vol. 34, no. 2, pp. 133–142, 2003.

[7] D. Lee, J. Villasenor, W. Luk, and P. Leong, "A hardware gaussian noise generator using the box-muller method and its error analysis," To appear in IEEE Transactions on Computers, 2006.

[8] G. L. Zhang, P. H. Leong, D.-U. Lee, J. D. Villasenor, R. C. Cheung, and W. Luk, "Ziggurat-based hardware gaussian random number generator," in *International Conference on Field Programmable Logic and Applications*. IEEE Computer Society Press, 2005, pp. 275–280.

[9] D.-U. Lee, W. Luk, J. D. Villasenor, G. Zhang, and P. H. Leong, "A hardware gaussian noise generator using the wallace method," *IEEE Transactions on VLSI Systems*, vol. 13, no. 8, pp. 911–920, 2005.

[10] M. George and P. Alfke, "Linear feedback shift registers in virtex devices," Application Note, Xilinx, Inc., Tech. Rep., 2001.

[11] B. Shackleford, M. Tanaka, R. J. Carter, and G. Snider, "FPGA implementation of neighborhood-of-four cellular automata random number generators," in *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, USA: ACM Press, 2002, pp. 106–112.

[12] D. B. Thomas and W. Luk, "High quality uniform random number generation through lut optimised linear recurrences," in *International Conference on Field-Programmable Technology*. IEEE Computer Society, 2005.

[13] D. B. Thomas and W. Luk, "Efficient hardware generation of random variates with arbitrary distributions," in *IEEE Symposium on FPGAs for Custom Computing Machines*, 2006.

[14] P. Kabal, "Generating gaussian pseudo-random deviates," Department of Electrical and Computer Engineering, McGill University, Tech. Rep., 2000.

[15] A. J. Walker, "An efficient method for generating discrete random variables with general distributions," *ACM Trans. Math. Software*, vol. 3, pp. 253–256, 1977.

[16] G. Marsaglia and W. W. Tsang, "The ziggurat method for generating random variables," *Journal of Statistical Software*, vol. 5, no. 8, pp. 1–7, 2000.

[17] C. S. Wallace, "Fast pseudorandom generators for normal and exponential variates," *ACM Transactions on Mathematical Software*, vol. 22, no. 1, pp. 119–127, 1996.