Non-uniform random number generation through piecewise linear approximations

D.B. Thomas and W. Luk

Abstract: A hardware architecture for non-uniform random number generation, which allows the generator's distribution to be modified at run-time without reconfiguration is presented. 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 typically around 1000 cycles to switch distributions at run-time. Comparison with Gaussian-specific generators shows 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. When higher statistical quality is required and multiple samples are required per cycle, a two-level piecewise generator can be used, reducing the RAM required per generated sample while retaining the simplicity and speed of the basic technique.

1 Introduction

Many computationally intensive applications have no closed form solution, and rely on Monte Carlo simulations or stochastic 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 because of 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.

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 are used to provide an empirical distribution for future events. Of particular interest is the behaviour of asset log-returns: if the price of an asset at time t is p_t, then the log-return at time t is $\ln(p_t/p_{t-1})$. The log-return takes into account the geometric scaling behaviour seen in stocks, treating an increase in value of 10% in two different stocks as equivalent, even if the stocks have very different prices.

Table 1 gives statistics for the log-returns of four different equities over the same time period, using daily closing prices adjusted for dividend payments [3]. A common approximation is to assume that the log-returns of equities are normally distributed, with both skewness (asymmetry) and kurtosis (pointiness) equal to zero, but this is clearly not the case: the skewness and kurtosis of the Microsoft log-returns show 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 become available.

This paper presents a new hardware architecture for non-uniform RNG 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;
- a means of increasing statistical quality and reducing RAM requirements when multiple identically distributed cycles are needed per cycle.

2 Background

The probability distribution of a continuous random variable X can be described using its 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

doi:10.1049/iet-cdt:20060188

Paper first received 1st November 2006 and in revised form 12th April 2007

The authors are with the Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2AZ, UK

E-mail: dt10@doc.ic.ac.uk

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

Stock	Mean	Std-Dev	Skewness	Kurtosis
General Electric Co.	0.0008	0.028	-0.16	0.6
Xilinx Inc.	0.0009	0.043	-0.09	1.5
Altera Corp.	0.0013	0.045	-0.19	2.5
Microsoft Corp.	0.0018	0.031	-0.59	10.3

of the distribution's probability density function (PDF) $f(x)$, which measures the point-wise probability of each value

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

Non-uniform RNGs often use two steps: uniform RNG, 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 $x_1, x_2, \ldots$ in the range [0, 1], and then applies the ICDF to give $y_i = F^{-1}(x_i)$. For example, the exponential distribution has $F(x) = 1 - e^{-x}$, allowing direct generation of exponential variates using $F^{-1}(x) = -\ln(1-x)$. However, there are many useful distributions for which there is no closed form solution for the ICDF (e.g. the normal distribution), so 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) RNGs, for example by using piecewise Hermite interpolation of the ICDF [4]. Another technique is to use the acceptance-rejection method, which generates candidate samples using a simple distribution, and then rejects those samples that do not also lie under the more complex target PDF, for example via the ratio-of-uniform-based generators [5]. These techniques can be used to approximate almost any distribution, but may require significant pre-processing before each new target distribution can be sampled. These methods also rely on 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 [6], or irregular domain partitioning to allow smaller tables [7]. Although these techniques can achieve a usable level of accuracy, both methods have some drawbacks. The regular domain partitioning scheme typically requires large off-chip RAMs to achieve sufficient distribution accuracy, whereas the introduction of an irregular domain partitioning limits the generator to a single class of distributions.

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 $\ln x$, $\sqrt{x}$ and $\sin x$. 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 [8]. 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 must be 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 [9]. The Wallace generator uses a novel approach to produce Gaussian samples directly, rather than transforming uniform samples. It has an efficient hardware implementation [10], 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) [11], 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. Nonlinear recurrences based on cellular automata (CA) have also been used in hardware [12], 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 [13]. Although designed to be efficiently implemented using word-based operations, such as the bit-wise logical and shift operations on 32-bit integers found in most CPU instruction sets, it is also usable in hardware [8, 9]. A more efficient (although more complex to implement) linear recurrence is also possible, which is optimised for lookup table (LUT)-based architectures [14]. These LUT-optimised linear recurrences provide n output bits per cycle, using a total of n LUTs and n flip-flops (FF), and have a period of $2^n - 1$, with good statistical quality.

3 Approximation algorithm

The non-uniform generation technique presented 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 PDF $t(x)$ that needs to be generated, we can calculate a set of n weights $w_1, \ldots, w_n \in [0, 1]$ and a set of component PDFs $g_1, \ldots, g_n$, such that the weighted combination $m(x)$ of the component PDFs equals the target PDF

$$t(x) \equiv m(x) = \sum_{i=1}^{n} w_i g_i(x), \quad \text{where} \sum_{i=1}^{n} w_i = 1 \qquad (2)$$

Similarly, the CDF $M(x)$ of the combination is the weighted sum of the component CDFs $G_a, \ldots, G_n$

$$T(x) \equiv M(x) = \sum_{i=1}^{n} w_i G_i(x) \qquad (3)$$

Assuming that some mechanism for generating samples from $g_1, \ldots, g_n$ is available, then samples from $t(x)$ can also generated, by randomly selecting one of the components according to their weights, then sampling the selected component distribution.

It is always possible to compute a weighted combination $m(x)$ that exactly matches $t(x)$ in this way, but for most distributions of interest this requires that at least one of the component distributions is as complex as the target PDF to calculate. However, if the weight of the complex component distribution can be reduced sufficiently, and the rest of the component distributions are very simple, then this can be a useful approach. For example, this is used in the Ziggurat method [15], where a rejection step is used to calculate the complex component distributions.

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 [16], 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.

Here 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 independent uniform samples together. The PDF of the sum of two independent variates can be obtained by convolving the PDFs of the two variates (see Section I.4 of [17]). Uniform variates have rectangular PDF functions, so the convolution of two independent identically distributed (IID) uniform variates results in a symmetric triangular PDF. This can be seen as a consequence of the central-limit theorem: addition of larger numbers of uniform variates would produce a PDF that gradually loses its triangular shape, converging on the Gaussian PDF as the number of uniform samples approaches infinity.

All the component triangles have the same width 2δ, and are spaced at increments of δ, that is the mean of component i is $i\delta + k$ (where k is an additive offset), so the PDF of the mixture is simply the linear interpolation of the triangle weights, and can be calculated exactly. Fig. 1 shows a simple example where components with $\delta = 0.5$ are used to approximate the Gaussian distribution (for clarity only

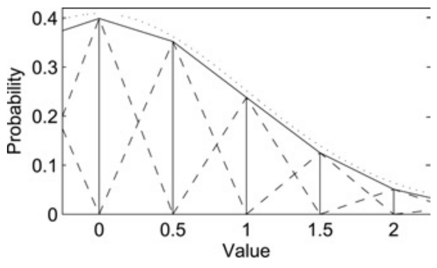

Fig. 1 *Simple approximation to the Gaussian distribution, using triangles with $\delta = 0.5$*

Note that only the positive half is shown, and the Gaussian PDF is offset vertically to distinguish between approximation and target PDF

a portion of the total PDF range is shown). The fine dotted line shows the smooth target Gaussian, whereas the dashed lines show the PDFs of the triangular components. The solid line just under the Gaussian PDF shows the PDF of the piecewise approximation (the Gaussian PDF has been offset vertically in the diagram to allow the two to be distinguished from each other).

Given a set of n triangles one should choose δ and k so as to provide good coverage of the 'interesting' part of the PDF range. So if the range $[a,b]$ needs to be covered using n triangles, one could choose $\delta = (b - a)/(n - 1)$ and $k = a - \delta$ to spread the triangles evenly across the range. In the case of an analytically defined target distribution with CDF $T(x)$ this range can be chosen to include a certain proportion of the probability mass, for example $a = T^{-1}(2^{-p})$, $b =^{-1} (1 - 2^{-p})$, with $p \gg 1$. The architecture and results presented in the implementation section take advantage of binary arithmetic for efficiency, and so use $\delta = 2^{-r}$ for integer r, and $k = 0$. As with many practical aspects of this type of generator, there is an application specific trade-off between efficiency and accuracy: in some applications it may be appropriate to use a non-binary power for δ, weighing the cost of a multiply per generated sample against an increase in overall accuracy.

A first approximation to the weights $w_1, \ldots, w_n$ can be provided by evaluating the target PDF at each triangle's centre, $w_i = t(i\delta + k)$, 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 attempt to find the best linear combination of triangles in the least-squares sense [18]. This method has been used in a similar approach to approximate the Gaussian distribution [19], but this approach was found to suffer from two problems.

First, when applied to highly skewed distributions, such as the exponential distribution, floating point calculation errors in the matrix inversion led to a completely unusable result. It is possible that a careful analysis of the problem might allow more accuracy to be returned, for example by using more robust algorithms such as singular value decomposition, but this still leaves the second problem of computational complexity. Solution of a dense system of equations is typically an $O(n^3)$ problem, and while it is feasible to solve the system for many thousands of triangles, it may take a number of minutes or hours, and require many megabytes of memory. Ideally it should be possible for a relatively modest embedded processor to approximate a new distribution, so a less computationally expensive approach is needed.

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 error with a granularity of $\delta/2$. The iterative algorithm used is

```
(1) Set $w_1, \ldots, w_n$ to the first approximation using $t(x)$.
(2) Set $c = i\delta/2 + k$, for $1 \le i \le 2n + 1$.
(3) Calculate $p = t(c)$, the vector of target probabilities.
(4) Calculate $d = p - m(c)$, 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. Note that the weights must not turn negative: for example. use the steps $w_i' = \max(0, w_i - e_i)$ and $e_i' = w_i - w_i'$ for the mid-point case.
(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: $w_j = e_i(1/2d_{2j-1} + d_{2j} + 1/2d_{2j+1}) + w_j, j = 1, \ldots, n$.
(9) If the solution does not meet the termination criteria return to Step 4.

The algorithm termination criteria can be defined in a number of ways, such as performing a fixed number of iterations, or waiting till the maximum or average error reaches a minimum point. The termination criterion used in this paper is to examine Σd (the total approximation error) on successive iterations, and to exit when the new value of Σd is greater than 0.9 times the previous value of Σd. Also, note that the approximation probability must drop to zero at the left and right sides of the first and last triangle, even if the target probability does not, so the algorithm implementation must handle these boundary conditions appropriately.

Fig. 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 (Note that this figure just demonstrates the rapid algorithm convergence: more rigorous tests of the output distribution quality are made in the evaluation section.) The Exchange [20], Football [21] and Microsoft [22] are all empirically defined, smoothed using the ksdensity command in Matlab [23]. Even after smoothing the empirical PDFs still exhibit local irregularities, and the optimiser cannot achieve the lower overall error achieved with the smoother analytically defined distributions.

It should be pointed out that there are no real theoretical underpinnings for this approximation algorithm: it is just a simple algorithm that the authors have found to work well in practise, and that has the advantage of being very fast. It is possible that with certain PDFs the algorithm may display instability and provide a very poor solution (although so far this has never been observed), and it is almost certain that it does not provide the 'optimal' approximation, for any definition of 'optimal'. However, the main goal of this paper is to

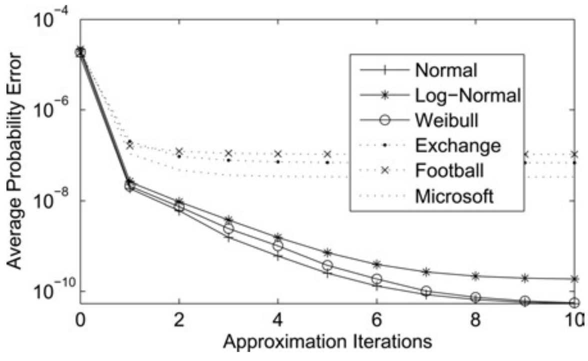

Fig. 2 *Average per-value approximation error (over* 2^{16} *output values), for approximations to analytic and empirically derived distributions*

provide a practical and efficient method for approximating arbitrary distributions, and with these goals in mind this simple algorithm is highly appropriate.

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 [24].

The alias method makes use of two tables, calculated in $O(n)$ time, one of which contains probability thresholds between zero and one $(t_1, \ldots, t_n)$, whereas the other contains alternate indices $(a_1, \ldots, a_n)$ in the range $1-n$. To sample from the discrete 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 y in the range $[0, 1]$ is then also generated. If $y < t_i$, then i is used as the selected sample from I, 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 size of a typical RAM implemented using contemporary block-RAM (such as the Xilinx RAMB16s used in the next section). 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 $w_1, \ldots, w_n$ 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; $t_1, \ldots, t_n$, probability thresholds between 0 and 1; and $a_1, \ldots, a_n$ 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 $c \leftarrow i\delta + k$, centre 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 within the selected triangle).

The next section examines the hardware implementation of this type of generator, followed in Section 5 by an evaluation of the statistical quality and a comparison with hardware techniques used for Gaussian random number generation. Section 6 then introduces a method that can be used to overcome the implicit trade-off between storage usage and statistical quality when multiple generators are used in a design.

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. The last point is not true in general, because of 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 exact value of x will be determined by the number of triangles available and the range of the distribution to be approximated, and to some extent depends on the fixed-point scale applied to the output of the generator.

The first step in translating the algorithm to hardware is to assign widths to the variables. The index variables ($a_1, \ldots, a_n$ and i) must have width $iw = \lceil \log_2 n \rceil$, as they address triangles $1, \ldots, n$. The threshold variables ($t_1, \ldots, t_n$ and y) are quantised to a 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.

In principle the value of tw should be incorporated into the approximation of the target distribution, but in practice it is easier to perform the approximation process in floating point, then quantise the weights to a value of tw that is convenient for the width of the available RAMs. No rigorous tests have been performed, but visual inspection of approximated PDFs and practical experience suggests that this quantisation has no effect for 'sensible' values of tw (e.g. the values of tw seen in Table 2).

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 according to the range of the target distribution.

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.

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

n	64		512		2048		1024
ow	12		16		24		16
tw	8	16	16	27	16	25	26
RAMs	–	–	05 (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

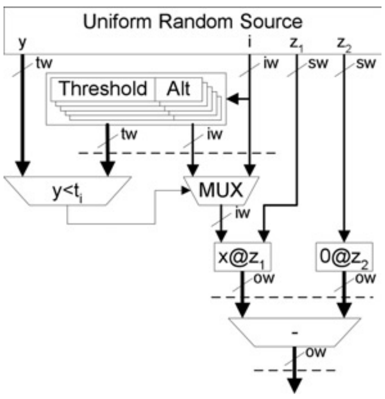

Fig. 3 *Hardware architecture of the generator algorithm*

Fig. 3 shows the architecture of the generator, including the widths of the signals (the @ operator represents bit-wise 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:

- a RAM containing n elements of width $tw + iw$;
- one tw bit comparator (tw LUTs);
- one iw multiplexer (iw LUTs);
- one ow bit subtracter ($iw + sw$ LUTs);
- a RNG capable of supplying $rw = tw + iw + 2sw$ independent random bits per cycle (tw LUTS using the method in [14]).

So in total the LUT usage can be estimated as $3iw + 2tw + 3sw$ LUTs.

The critical path will be from the random index i, through the RAM, into comparator, through the MUX, and 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 Fig. 3 as 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. For example, i must be pipelined before entering the MUX as it is used to generate the RAM address, but y, z_1 and z_2 do not need to be delayed.

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.

An interesting feature of this generator is that increasing the number of output bits does not significantly increase the resource usage of the generator. For example, with $iw = 10$, $tw = 26$ and $ow = 16$ approximately 136 LUTs are needed, but increasing the output width to $ow = 32$ and 64 only increases the estimated LUT count to 184 and 280, respectively (without requiring any increase in RAM size). Techniques that are based on polynomial evaluation require significantly more area and RAM as the output width increases, because of the need to support wider multipliers and table constants. Although the increase in fractional bits does not improve the overall goodness of fit of the distribution, it is useful in applications which assume that the distributions are continuous, as it reduces the probability of duplicate values. More fractional bits are also useful when converting samples to floating-point, as it allows values closer to zero to be produced. This is particularly important when dealing with leptokurtic (pointy) distributions centred around zero, such as the log-return distributions of assets.

5 Evaluation

Table 2 shows the performance of a selection of generators in the Virtex-II architecture, generated through a single parametrised 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 $s = 2^4, 2^5, 2^6, \ldots, 2^{36}$ 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, because of 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.

A second quality metric is the statistical randomness of the output, relating to the lack of correlations in the output stream, rather than the shape of the distribution over many samples. Non-random behaviour is usually detected by generating a large stream of samples, then looking for patterns that are unlikely to occur in a random stream, such as long sequences of very small values that would be unlikely to occur by chance. These tests are collected into test batteries such as Diehard [25] and Crush [26], which give an indication of whether a test looks random. However, these tests usually assume that the random stream is from the uniform distribution, whereas the generators presented here are designed to produce non-uniform distribution.

A strong supporting argument for the quality of the non-uniform generators is that the underlying uniform generators have good statistical quality. The last generator in Table 2 uses a 96-bit linear recurrence (to take advantage of RAM sharing the 48-bit sources for a pair of generators are extracted from a single shared generator). According to the tests performed in [14], a 96-bit generator passes all the tests in Diehard, and only fails ten (out of 96) tests in Crush. Of the ten failed tests nine are specifically related to linear-complexity (a defect shared by all binary linear recurrences such as LFSRs and combined Tausworthe generators), which is likely to be masked when the two uniform components are added together to create a sample from the triangular distribution.

Applying test batteries to the generators directly requires conversion of the output stream to the uniform distribution by using the distribution CDF. The Diehard and Crush tests were applied to the high-performance Gaussian generator from the last column of Table 2. All tests were passed except for a class of test called Max-of-T, which looks at the distribution of the maximum sample within successive t-tuples. The Max-of-T test is applied four times with different parameters in the Crush battery and all four were failed because the approximation to the Gaussian PDF is poor in the tails (i.e. for high and low values in the range). This is one of the reasons that the generator fails the χ^2 test for 2^{36} samples. When the test batteries are applied to the high quality 2048 triangle generator (which passed the χ^2 test), all the tests in Crush and Diehard are passed, suggesting that the only statistical defects are those because of poor approximation of the target distribution, rather than from correlations within the output stream.

Table 3 provides a comparison between the performance of the speed-optimised piecewise generator from Table 2 and other types of Gaussian RNG, 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 (the number of cycles taken to change distributions is simply the time taken to reprogram the generator's RAM, in this case a 1024 element RAM). All the other generators are limited to just the standard Gaussian distribution, and require extra logic even to change the distribution variance.

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

Class	Previous hardware methods				Piecewise	Software	
Method	Box-Muller	Wallace	Ziggurat	Trapezoid	$n = 2^{10}$, $tw = 26$	Ziggurat	Wallace
Reference	[8]	[10]	[9]	[16]	–	[15]	[27]
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}	–	–

6 Two-level approximations

A drawback of the piecewise approximation methods is that large RAMs are needed to achieve high-quality approximations. Providing a good local approximation to the PDF requires a high density of piecewise segments to match the underlying curve, whereas providing coverage into the PDF tails requires a large area of the range to be covered with triangles. These two factors mean that, as a rule of thumb, to double the quality of the generator (in terms of samples that can be produced before statistical failure), the size of the RAM must be approximately doubled.

Although it is possible to create large RAMs by combining multiple block-RAMs, this is not an efficient use of available resources, as only one entry from the RAM is used per-cycle so much of the potential block-RAM bandwidth is wasted. Where a range of block-RAM sizes are available it may be possible to choose the largest size, for example the Stratix-II M-RAM block can provide 2^{14} 36-bit entries, which is likely to be large enough for the vast majority of quality requirements. However, such large RAMs are scarce resources, and even the largest Stratix-II only has nine available, severely limiting the number of RNGs that can be placed in a design. Neither are off-chip RAMs a solution, as these are also scarce and valuable resources, and may also limit the maximum clock speed that generators can reach because of the need to do a RAM lookup in every cycle.

One solution to this problem is to split the generation process into two generators: a primary generator that uses a small number of triangles (and hence a small RAM) to provide a coarse approximation to the target PDF, and a residual generator that uses a much larger RAM to provide a fine-grain 'fix-up'. The advantage is that the two generators can be arranged so that the output of the primary is used in the vast majority of cycles, while only in the small number of remaining cycles is the output of the residual used. This allows the residual generator (and most importantly, the residual generator's valuable RAM), to be shared among multiple generators (all of which must be producing the same distribution).

A disadvantage is that sharing the residual generator introduces a resource conflict, as the residual generator's fixed output rate of one residual sample per second cannot be guaranteed to meet bursts in demand from the set of primary generators. This means that there is a possibility on each cycle that one or more of the generators may not produce a valid output. Fortunately many pipelined simulation architectures are able to deal with unavailable resources on any given cycle, simply by not updating the simulation state when it cannot be fully calculated. For example, in the architecture presented in [28], this is used to accommodate the situation where there is no space in the output channel. However, it is necessary to make the probability of a generator-induced pipeline stall as low as possible if performance is not to be affected.

For a two-level generator to work, it is necessary for the distributions of the primary and residual generators to be carefully designed, such that the combination of the two produces the target distribution. Fig. 4 shows the stages in this process for a highly simplified example. Part (a) shows the curved target PDF, and a very coarse two-triangle piecewise approximation to the curve. Note that the coarse approximation stays entirely underneath the target PDF, unlike previous approximations where the approximation's PDF was sometimes above, and sometimes below. The gap between the target and coarse approximation forms a residual distribution, shown in (b). This residual distribution

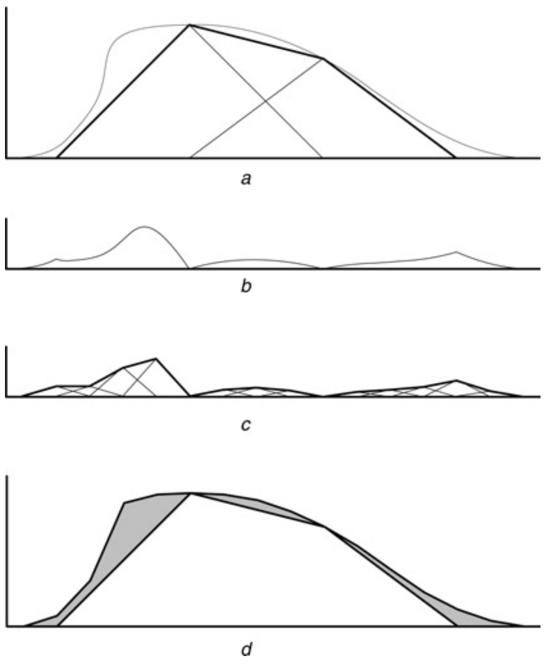

Fig. 4 *Two-level approximation strategy*

a Very coarse-grain approximation to the target PDF is made
b Leaving a residual distribution
c Fine-grain approximation to the residual distribution is made
d Combined with the coarse-grain approximation

can then be approximated using a fine-grained piecewise approximation, shown in (c). If the coarse-grained and fine-grained distributions are combined as in (d), then the result is a good approximation to the target distribution. The relative areas of the primary and residual distribution (the interior white and surrounding grey area, respectively) determine the probability of sampling from each generator.

If m_p and m_r are the PDFs of the piecewise generators chosen for the primary and residual distributions, then the overall generated PDF is given by

$$m(x) = a_p m_p(x) + (1 - a_p) m_r(x) \quad (4)$$

The constant a_p represents the proportion of cycles where the primary generator is used, and should be as high as possible to reduce load on the secondary generator. Any two piecewise distributions can be chosen for m_p and m_r, as long as the overall $m(x)$ is a good approximation to the target PDF. In practise, the number of triangles in m_p and m_r will be largely determined by the size and number of the different types of RAM available. For example, in a Stratix-II device with TriMatrix memory [29], the primary generator RAMs might be placed in the small but numerous M4K blocks, whereas the residual generator's RAM might be placed in one of the larger but more scarce M-RAM blocks, suggesting 128 triangles for the primary generators and 2^{14} triangles for the residual generator. In an architecture with a single RAM type, such as Virtex-II [30], it will be necessary to either build a larger RAM for the residual generator out of any block RAMs that can be spared, or use an external RAM.

There are many possible ways of choosing the distributions of m_p and m_r, with one of the simplest being the approach used in Fig. 4, where first a combination of a_p and m_p is chosen such that the primary lies completely under $t(x)$, the target PDF. This is easily achieved by using the standard method for fitting the piecewise PDF, then by scaling the height of the two triangles on either side of piecewise segments that extend above the target

PDF. The area of the piecewise function after scaling determines a_p, and then scaling all the triangle heights by $1/a_p$ provides the piecewise PDF m_p. The residual piecewise PDF can then be chosen using the standard approximation method, fitting to the residual target PDF

$$t_r(x) = \frac{t(x) - a_p m_p(x)}{(1 - a_p)} \tag{5}$$

The value of a_p that can be achieved is dependent on both the number of triangles used and the characteristics of the target distribution. The reason a_p is so important is that it determines the number of primary generators that can be paired with a single residual generator. If the residual generator runs at the same speed as the primary generators, then it can supply residual samples for at most $\lfloor 1/(1 - a_p) \rfloor$ primaries.

Fig. 5 shows a_r (where $a_r = 1 - a_p$, the probability of using the residual generator on each cycle, i.e. a miss) for a number of such combinations. Both the normal and Weibull distributions use the primary PDF over 99% of the time when 512 triangles are used (equivalent to using a single Virtex-II block-RAM for the primary generator), whereas the log-normal can be used in more than 97% of cycles. This suggests that a could be shared among up to 40 RNG primary generators, at least for these distributions. However, the means by which residual samples are distributed to primary generators could reduce this maximum figure significantly.

Fig. 6 shows a concrete architecture for managing the distribution of residual samples, using a chain of registers. This chain ensures that each residual sample is only used once, by maintaining a tag bit along with each sample indicating whether the sample is valid. There is also a tagged register associated with each primary generator, which can be fed by a paired register in the chain. In each cycle, any registers with invalid tags will take the sample from the previous register in the chain. If a register has a valid sample, and the next register in the chain also has a sample, then the register will check the tag of the paired primary generators' sample, giving the sample to the primary generator if it needs it. When a primary generator needs a residual sample it takes it from the local register, marking the sample as invalid. This invalid token then forms a bubble in the chain of valid samples, which will move backwards up the chain as new valid samples move downwards.

The operation of each register is determined purely by its own tag bits and those of its immediate neighbours, allowing each register stage to be implemented in $ow + 1$ register bits (where ow is the output width of the generator) and two 4-LUTs. Using a register chain has the advantage that in

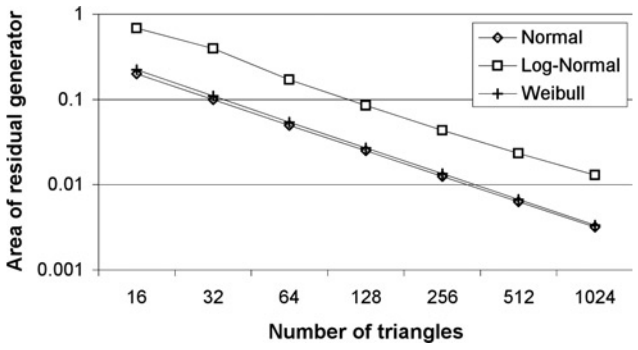

Fig. 5 *Area of the residual PDF when using different numbers of triangles in the primary generator, shown for three different distributions*

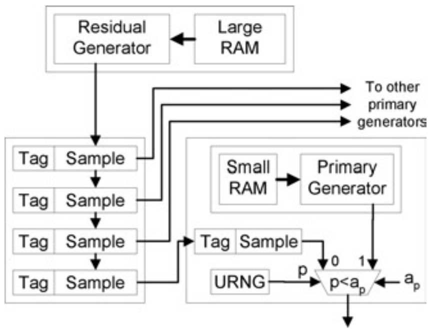

Fig. 6 *Distribution of residual samples to primary generators through a register chain*

large designs the arbitration and communication happen at a local level, requiring no large-fanout control or data nets. This is critical when samples are being distributed from a single residual generator out to a large number of primary generators, particularly at high clock-rates.

The distribution chain cannot guarantee that a residual sample will be available to a primary generator in each cycle, as the nature of the proposed architecture only allows a primary generator to receive a new sample every other cycle (and even then only if there is a valid sample in the associated shift register). However, the probability of a primary generator requiring two residual samples in a row is a_r^2, which even for very low values of a_r will happen relatively often at typical FPGA clock frequencies. If no residual sample is available then the primary generator must stall the process it is supplying, and wait for a residual sample to arrive. The probability of these stalls needs to be as low as possible if overall performance is to be maintained, and can be derived for different parameters through simulation.

Fig. 7 shows the probability of each primary generator stalling per cycle as the number of primary generators (n) being supplied increases, using values of $a_r = \{2^{-3}, 2^{-4}, 2^{-5}\}$. In all cases, the stall probability gradually increases from 0 when $n = 1$, as there is one residual generator per primary, up to an asymptotic limit of a_r as $n \leftarrow \infty$, when there are so many primary generators that the residual generator makes no difference. The overall performance can be measured as the stall probability times

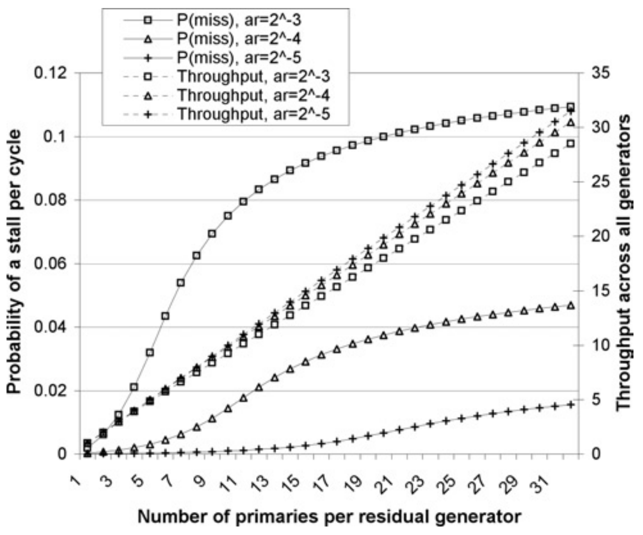

Fig. 7 *Probability of a generator stalling per cycle for different residual areas (a_r) as the number of primaries is increased, and the resulting overall throughput across all primary generators*

Table 4: Area and performance for a two-level generator in the Virtex-II architecture, using a residual generator with eight RAMs feeding different numbers of primary generators

		Absolute performance					Relative performance		
	Pr (stall)	RAMs	LUTs	FFs	Slices	MHz	RAMs	Slices	Throughput
Residual	0	8	212	301	168	247	1.00	1.00	1.00
Distributor	–	0	38	82	47	270	–	–	–
Primary	0	0.5	190	205	120	259	–	–	–
$n = 1$	4.89×10^{-7}	9	441	589	337	245	1.13	2.01	0.99
$n = 2$	1.21×10^{-6}	9	669	879	505	245	1.13	3.01	1.98
$n = 4$	2.83×10^{-6}	10	1125	1454	840	241	1.25	5.00	3.90
$n = 8$	6.28×10^{-6}	12	2051	2600	1511	237	1.50	8.99	7.68
$n = 16$	1.51×10^{-5}	16	3873	4921	2865	230	2.00	17.05	14.90
$n = 32$	4.62×10^{-5}	24	7509	9542	5514	233	3.00	32.82	30.18
Wallace, $n = 1$	–	7	–	–	895	155	1.00	1.00	1.00
Wallace, $n = 16$	–	112	–	–	14 359	115	16.00	16.04	11.80

the number of generators, giving the average number of cycles generated per second. The graph shows that performance increases approximately linearly with the number of generators, with different slopes for the different values of a_r. With the lowest value shown of $a_r = 2^{-5}$, the performance with 32 generators is 31.5, providing 98.5% efficiency.

The top sections of Table 4 shows the results of implementing a two-level piecewise approximation in the Virtex-II architecture, generating 24 bit samples, and using 4096 and 512 triangles in the residual and primary generators, respectively (corresponding to 8 and 1/2 block RAMs). A residual generator with 4096 triangles was found sufficient to generate more than 2^{38} samples from the normal distribution without failing the χ^2 test, and provides PDF coverage in the range of $[-8, +8]$, so can be considered to be of high enough quality for all but the most demanding and long-running simulations. The first three rows show the performance for the three components, where the distributor represents the extra logic that must be added per-primary generator to integrate it into the two-level architecture (including resources used in the distribution chain). Below these rows the performance for two-level generators with different numbers of primaries are shown, including the probability of stalling (assuming a_r is that of the normal distribution). The changes in resource usage and throughput are highlighted in the columns on the right, showing performance relative to the residual generator (which is capable of generating the target distribution by itself). Although the number of slices doubles each time the number of primaries double, the number of RAMs grows at a much slower rate, so 32 primaries only use three times the number of RAMs of a single residual generator.

For comparison, the bottom section of the table shows results from an experiment performed in [10], where the resource usage and performance were monitored as parallel generators are added to a design. Here the number of RAMs is simply a multiple of the number used by one generator, so 112 RAMs are required for $n = 16$, compared to 16 for the two-level generator. The clock rate also degrades much faster, which can be attributed to the much higher per-node complexity of the Wallace generator, as each generator needs to route to and from seven block-RAMs and four block-multipliers, and each generator is relatively large. By comparison, the per-node complexity of the two-level generator is low, with each generator only requiring read access to one block-RAM, and the shift-chain is designed to use only local communications. This allows the clock frequency to stay relatively high, achieving a per-generator relative performance of 93% when comparing $n = 1$ against $n = 16$, while the Wallace degrades to 74%.

The two-level approach provides a valuable means of increasing quality and reducing RAM usage while retaining the simplicity and efficiency of the piecewise approximation method, but as presented can only be used under certain conditions. The main constraint is that the process consuming the random numbers must be able to stall in cycles where the generator does not produce an output, which is possible in many simulation applications, but may add too much overhead in some architectures, or in certain real-time tasks may be impossible. The second constraint is that all the primary generators sharing a residual generator must be generating the same distribution. This is often the case where a large number of identical simulation pipelines operate in parallel, as each pipeline will require independent random samples from the same distributions.

7 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 occur without reconfiguration.
- 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.
- Ability to scale to large numbers of generators: using a two-level approximation, multiple high-quality independent identically distributed samples can be produced per cycle, without requiring large numbers of RAM resources.

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. Although 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 basic 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. In situations where a large number of high-quality IID samples are required, for example when multiple identical simulation pipelines are placed in a large device, the two-level technique allows the number of samples per cycle to be scaled up, without requiring a large number of RAMs.

8 References

1 Zhang, G.L., Leong, P.H.W., Ho, C.H., Tsoi, K.H., Lee, D.-U., Cheung, R.C.C., and Luk, W.: ‘Reconfigurable acceleration for Monte Carlo based financial simulation’. Proc. Int. Conf. Field-Programmable Technology, 2005, IEEE Computer Society Press, pp. 215–224

2 Negoi, A., and Zimmermann, J.: ‘Monte Carlo hardware simulator for electron dynamics in semiconductors’. Int. Annual Semiconductor Conf., Sinaia, Romania, 1996, pp. 557–560

3 Yahoo! finance, http://finance.yahoo.com, daily closing data as of 24th of March 2007

4 Hörmann, W., and Leydold, J.: ‘Continuous random variate generation by fast numerical inversion’, *ACM Trans. Modeling Comput. Simulation*, 2003, **13**, (4), pp. 347–362

5 Leydold, J.: ‘Short universal generators via generalized ratio-of-uniforms method’, *Math. Comp.*, 2003, **72**, (243), pp. 1453–1471

6 McCollum, J.M., Lancaster, J.M., Bouldin, D.W., and Peterson, G.D.: ‘Hardware acceleration of pseudo-random number generation for simulation applications’. IEEE Southeastern Symp. on System Theory, 2003, pp. 299–303

7 Boutillon, E., Danger, J.-L., and Ghazel, A.: ‘Design of high speed AWGN communication channel emulator’, *Analog Integrated Circuits and Signal Process.*, 2003, **34**, (2), pp. 133–142

8 Lee, D., Villasenor, J., Luk, W., and Leong, P.: ‘A hardware gaussian noise generator using the box-muller method and its error analysis’, *IEEE Trans. Comput.*, 2006, **55**, (6), pp. 659–671

9 Zhang, G.L., Leong, P.H., Lee, D.-U., Villasenor, J.D., Cheung, R.C., and Luk, W.: ‘Ziggurat-based hardware gaussian random number generator’. Proc. Int. Conf. Field Programmable Logic and Applications, IEEE Computer Society Press, 2005, pp. 275–280

10 Lee, D.-U., Luk, W., Villasenor, J.D., Zhang, G., and Leong, P.H.: ‘A hardware gaussian noise generator using the wallace method’, *IEEE Trans. VLSI Syst.*, 2005, **13**, (8), pp. 911–920

11 George, M., and Alfke, P.: ‘Linear feedback shift registers in Virtex devices’, 2001, Application Note Xilinx Inc., Tech. Rep., 2001

12 Shackleford, B., Tanaka, M., Carter, R.J., and Snider, G.: ‘FPGA implementation of neighborhood-of-four cellular automata random number generators’. Proc. ACM/SIGDA Int. Sym. Field-Programmable Gate Arrays, New York, USA, ACM Press, 2002, pp. 106–112

13 L’Ecuyer, P.: ‘Maximally equidistributed combined tausworthe generators’, *Math. Comp.*, 1996, **65**, (213), pp. 203–213 [Online]. Available: citeseer.ist.psu.edu/25662.html

14 Thomas, D.B., and Luk, W.: ‘High quality uniform random number generation through lut optimised linear recurrences’. Proc. Int. Conf. Field-Programmable Technology, IEEE Computer Society, 2005

15 Marsaglia, G., and Tsang, W.W.: ‘The ziggurat method for generating random variables’, *Journal of Statistical Software*, 2000, **5**, (8), pp. 1–7

16 Thomas, D.B., and Luk, W.: ‘Efficient hardware generation of random variates with arbitrary distributions’. Proc. IEEE Sym. on FPGAs for Custom Computing Machines, 2006

17 Devroye, L.: ‘Non-uniform random variate generation’ (Springer-Verlag, New York, 1996)

18 Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P.: ‘Numerical recipes’ (Cambridge University Press, 1997, 2nd edn.)

19 Kabal, P.: ‘Generating gaussian pseudo-random deviates’, Department of Electrical and Computer Engineering, McGill University, Tech. Rep., 2000

20 Weigend, A.S., Huberman, B.A., and Rumelhart, D.E.: ‘Predicting sunspots and exchange rates with connectionist networks’ in ‘Nonlinear modeling and forecasting’ (Addison-Wesley, 1992), pp. 395–432

21 UEFA, ‘European Cup: points scored’, www.european-footballstatistics.co.uk, 2006

22 finance.yahoo.com. ‘Microsoft daily close price (log-returns)’, march 1986 to march 2006

23 Bowman, A., and Azzalini, A.: ‘Applied smoothing techniques for data analysis’ (Oxford University Press, 1997)

24 Walker, A.J.: ‘An efficient method for generating discrete random variables with general distributions’, *ACM Trans. Math. Softw.*, 1977, **3**, pp. 253–256

25 Marsaglia, G.: ‘The Diehard random number test suite’, http://stat.fsu.edu/pub/diehard/, 1997

26 L’Ecuyer, P., and Simard, R.: ‘TestU01 random number test suite’, http://www.iro.umontreal.ca/simardr/indexe.html

27 Wallace, C.S.: ‘Fast pseudorandom generators for normal and exponential variates’, *ACM Trans. Math. Software*, 1996, **22**, (1), pp. 119–127

28 Bower, J.A., Thomas, D.B., Luk, W., and Mencer, O.: ‘A reconfigurable simulation framework for financial computation’. Proc. ReConFig. ’06, 2006

29 ‘Stratix II Device Handbook, Volume 1’, Altera Corporation, 2005

30 ‘Virtex-II Platform FPGAs: Complete Data Sheet’, Xilinx, Inc., 2000