2006-01

Programmable numerical function generators based on quadratic approximation: Architecture and synthesis method

Programmable Numerical Function Generators Based on Quadratic Approximation: Architecture and Synthesis Method

Shinobu Nagayama
Dept. of CE
Hiroshima City Univ.
Hiroshima 731-3194, Japan

Tsutomu Sasao
Dept. of CSE
Kyushu Inst. of Tech.
Iizuka 820-8502, Japan

Jon T. Butler
Dept. of ECE
Naval Postgraduate School
CA 93943-5121, USA

Abstract— This paper presents an architecture and a synthesis method for programmable numerical function generators (NFGs) for trigonometric, logarithmic, square root, and reciprocal functions. Our NFG partitions a given domain of the function into non-uniform segments using an LUT cascade, and approximates the given function by a quadratic polynomial for each segment. Thus, we can implement fast and compact NFGs for a wide range of functions. Implementation results on an FPGA show that: 1) our NFGs require only 4% **of the memory needed by NFGs based on the linear approximation with non-uniform segmentation; and 2) our NFGs require only** 22% **of the memory needed by NFGs based on the 5th-order approximation with uniform segmentation. Our automatic synthesis system generates such compact NFGs quickly.**

I. Introduction

Numerical function generators (NFGs) are often used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics, etc. The functions realized include trigonometric, logarithmic, square root, and reciprocal functions. High-performance CPUs usually have numerical coprocessors. However, embedded CPUs and CPUs on FPGAs do not have such coprocessors. Thus, FPGA implementation of numerical functions $f(x)$ is needed. Implementation by a single lookup table for $f(x)$ is simple and fast. For low-precision computations of $f(x)$ (e.g. x and $f(x)$ have 8 bits), this implementation is straightforward. For high-precision computations, however, the single lookup table implementation is impractical due to the huge table size. For such applications, the CORDIC (COordinate Rotation DIgital Computer) algorithm [1, 21] has been often used. Although CORDIC is implemented with compact hardware, it is iterative and therefore slow. For numerically intensive applications, faster evaluation of numerical function is required.

For fast evaluation of numerical functions, polynomial approximations have been used [9, 10, 19, 20]. These methods approximate the given numerical functions by piecewise polynomials, and realize the polynomials with hardware. Linear or quadratic approximations offer fast and relatively high-precision evaluation of numerical functions. However, the methods proposed so far are ad-hoc and not systematic. This paper proposes an architecture and a systematic synthesis method for NFGs based on quadratic approximation. By using the LUT cascade [8], many numerical functions are efficiently approximated by piecewise quadratic functions. Our synthesis method can be automated, so that fast and compact NFGs can be produced by non-experts. Fig. 1 shows the synthesis flow for the NFG. It converts the Design Specification described by Scilab [18], a MATLAB-like software, into HDL code. The Design Specification consists of a function $f(x)$, a domain for x, and an acceptable error. This system first partitions the domain into segments, and then approximates $f(x)$ by a quadratic function for each segment. Next, it analyzes the errors, and derives the necessary precision for computing units in the NFG. Then, it generates HDL code to be mapped into an FPGA using an FPGA vendor tool. Due to the page limitation, the error analysis for our NFGs is omitted here, but it is available in [14]. This paper extends [17] to quadratic approximations.

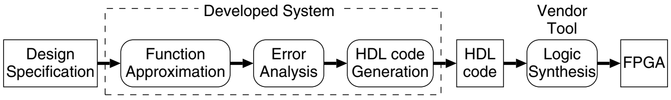


Fig. 1. Synthesis flow for NFGs.

II. Preliminaries

Definition 2.1 *The* ***binary fixed-point representation*** *of a value r has the form*

$$d_{n\_int-1}\ d_{n\_int-2}\ \ldots\ d_1\ d_0.\ d_{-1}\ d_{-2}\ \ldots\ d_{-n\_frac}, \tag{1}$$

where $d_i \in \{0,1\}$, n_int is the number of bits for the integer part, and n_frac is the number of bits for the fractional part of r. The representation in (1) is two's complement, and so

$$r = -2^{n\_int-1}d_{n\_int-1} + \sum_{i=-n\_frac}^{n\_int-2} 2^i d_i.$$

Definition 2.2 ***Error*** *is the absolute difference between the original value and the approximated value.* ***Approximation error*** *is the error caused by a function approximation, and* ***rounding error*** *is the error caused by a binary fixed-point representation.* ***Acceptable error*** *is the maximum error that an NFG may assume.* ***Acceptable approximation error (AAE)*** *is the maximum approximation error that a function approximation may assume.*

0-7803-9451-8/06/$20.00 ©2006 IEEE.

Definition 2.3 ***Precision*** *is the total number of bits for a binary fixed-point representation. Specially,* ***n-bit precision*** *specifies that n bits are used to represent the number; that is,* $n = n\_int + n\_frac$. *An* ***n-bit precision NFG*** *has an n-bit input.*

Definition 2.4 ***Accuracy*** *is the number of bits in the fractional part of a binary fixed-point representation. Specially,* ***m-bit accuracy*** *specifies that m bits are used to represent the fractional part of the number; that is,* $m = n\_frac$. *An* ***m-bit accuracy NFG*** *is an NFG with m-bit fractional part of the input, m-bit fractional part of the output, and a* 2^{-m} *acceptable error.*

III. Quadratic Approximation Algorithm

To approximate the numerical function $f(x)$ using quadratic functions, first, we partition the domain for x into segments. For each segment, we approximate $f(x)$ using a quadratic function $g(x) = c_2x^2 + c_1x + c_0$. In this case, the approximation error depends on the segmentation method and the values of coefficients c_2, c_1, and c_0 in the approximation polynomial.

For piecewise polynomial approximations, in many cases, the domain is partitioned into uniform segments [2, 6, 19]. Such methods are simple and fast, but for some kinds of numerical functions, too many segments are required, resulting in large memory.

For a given error, non-uniform segmentation of the domain uses fewer segments than the uniform segmentation [9, 17]. However, a non-uniform segmentation often requires a complicated segment index encoder (see Section IV), and results in larger and slower NFGs. To overcome this problem, a special non-uniform segmentation has been proposed [9]. This method produces a simple segment index encoder by restricting the segmentation points, and results in fewer segments as well as faster and more compact NFGs than produced by uniform segmentation. However, it is ad-hoc and non-optimum for the given function. Our NFG can implement any non-uniform segmentation with a fast and compact segment index encoder by using an LUT cascade [17] with a synthesis method that can be automated.

Selection of the approximation polynomial influences the number of non-uniform segments as well as the approximation error. In this paper, we use the 2nd-order Chebyshev approximation to approximate $f(x)$ with fewer non-uniform segments, and compute the approximated value. Since coefficients of the Chebyshev approximation polynomial are easily computed, it is suitable for automatic synthesis.

A. Segmentation Algorithm

For a segment $[s,e]$ of $f(x)$, the maximum approximation error $\varepsilon_2(s,e)$ of the 2nd-order Chebyshev approximation [11] is given by

$$\varepsilon_2(s,e) = \frac{(e-s)^3}{192} \max_{s \le x \le e} |f^{(3)}(x)|, \tag{2}$$

where $f^{(3)}$ is the 3rd-order derivative of f. From (2), $\varepsilon_2(s,e)$ is a monotone increasing function of segment width $e-s$. Using this property, we partition a domain into as wide segments as possible such that the approximation error is less than the specified error. Fig. 2 shows the non-uniform segmentation algorithm. The inputs for this algorithm are a numerical function $f(x)$, a domain $[a,b]$ for x, and an acceptable approximation error ε. Then, this algorithm approximates $f(x)$ with the acceptable approximation error ε, and produces t segments $[s_0,e_0],[s_1,e_1],\ldots,[s_{t-1},e_{t-1}]$. For step 2 in Fig. 2, the accurate computation of the value p where $\varepsilon_2(s_i,p) = \varepsilon$ is difficult. Thus, we obtain the maximum value p' satisfying $\varepsilon_2(s_i,p') \le \varepsilon$. Such p' can be found by scanning values of n-bit input x. However, it requires $O(2^n)$ search, and is time-consuming. Therefore, we compute the maximum value p' by setting 0 or 1 from MSB to LSB of x such that $\varepsilon_2(s_i,p') \le \varepsilon$. This requires $O(n)$ search. In the computation of $\varepsilon_2(s_i,p')$, the value of $\max_{s_i \le x \le p'} |f^{(3)}(x)|$ is computed by the nonlinear programming algorithm, which is one of the most efficient [7].

```
Input:    Numerical function $f(x)$, Domain $[a,b]$ for $x$,
          Acceptable approximation error $\varepsilon$.
Output:   Segments $[s_0,e_0],[s_1,e_1],\ldots,[s_{t-1},e_{t-1}]$.

Process:
   1.     Let $s_0 = a$ and $i = 0$.
   2.     Find a value $p$ $(\ge s_i)$ where $\varepsilon_2(s_i,p) = \varepsilon$.
   3.     If $p > b$, then let $p = b$.
   4.     Let $e_i = p$ and $i = i+1$.
   5.     If $p = b$, then let $t = i$, and stop the process.
   6.     Else, let $s_i = p$, and go to step 2.
```

Fig. 2. Non-uniform segmentation algorithm for the domain.

B. Computation of Approximate value

For each $[s_i,e_i]$, $f(x)$ is approximated by the corresponding quadratic function $g_i(x)$. That is, the approximated value y of $f(x)$ is computed as follows:

$$y = g_i(x) = c_{2i}x^2 + c_{1i}x + c_{0i}, \tag{3}$$

where the coefficients c_{2i}, c_{1i}, and c_{0i} are derived from the 2nd-order Chebyshev approximation polynomial [11]. Substituting $x - q_i + q_i$ for x in (3) yields the transformation

$$\begin{aligned} g_i(x) &= c_{2i}(x-q_i)^2 + (c_{1i} + 2c_{2i}q_i)(x-q_i) \\ &\quad + c_{0i} + c_{1i}q_i + c_{2i}q_i^2. \end{aligned} \tag{4}$$

In (4), let $c'_{1i} = c_{1i} + 2c_{2i}q_i$ and $c'_{0i} = c_{0i} + c_{1i}q_i + c_{2i}q_i^2$. Then, we have

$$g_i(x) = c_{2i}(x-q_i)^2 + c'_{1i}(x-q_i) + c'_{0i}. \tag{5}$$

This transformation reduces the multiplier size.

IV. Architecture for NFGs

Fig. 3 shows the architecture that realizes (5). It uses 7 units: the segment index encoder that computes the index i for segment $[s_i,e_i]$ including the input value x; the coefficients table for $-q_i$, c_{2i}, c'_{1i}, and c'_{0i}; the adder for $x + (-q_i)$; the squaring unit; two multipliers; and the final adder.

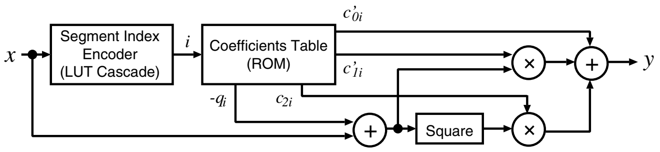
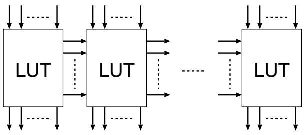


Fig. 3. Architecture for NFGs.

Interval	Index
$s_0 \leq x \leq e_0$	0
$s_1 < x \leq e_1$	1
$\vdots$	$\vdots$
$s_{t-1} < x \leq e_{t-1}$	$t-1$

(a) Segment index function.



(b) LUT cascade.

Fig. 4. Segment index encoder.

A *segment index encoder* converts x into a segment index i. It realizes the segment index function $seg\_func(x) : B^n \rightarrow \{0,1,\ldots,t-1\}$ shown in Fig. 4 (a), where x has n bits, $B = \{0,1\}$, and t denotes the number of segments. In [9], to simplify the segment index encoder, the values of s_i and e_i are restricted to what can be produced by a simple combinational logic circuit. Such a segmentation method results in many segments since it does not adapt to the given function. Our synthesis system uses the *LUT cascade* [8, 15, 16] shown in Fig. 4 (b) to realize arbitrary $seg\_func(x)$. It can be designed by functional decomposition using BDDs (Binary Decision Diagrams) representing $seg\_func(x)$. Our synthesis system uses a nonrestrictive segmentation. It is suitable for automatic synthesis. In LUT cascades, the interconnecting lines between adjacent LUTs are called *rails*. The size of an LUT cascade depends on the number of rails. The next theorem shows that the segment index functions are realized by compact LUT cascades.

Theorem 4.1 *[16] Let $seg\_func(x)$ be a segment index function with t segments. Then, there exists an LUT cascade for $seg\_func(x)$ with at most $\lceil \log_2 t \rceil$ rails.*

Our synthesis system uses heterogeneous MDDs (Multi-valued Decision Diagrams) [13] to find compact LUT cascades. Since the LUT cascade is suitable for the pipeline processing, it offers a fast and compact circuit. In Section VI, we will show that our architecture produces fast and compact NFGs for various numerical functions.

V. Implementation with FPGA

Modern FPGAs consist of logic elements (LEs) or configurable logic blocks (CLBs), synchronous memory blocks, multipliers (DSP units), etc. Our synthesis system efficiently generates NFGs using these components. Each unit for the NFG shown in Fig. 3 is implemented by the following components in an FPGA: 1) Segment index encoder (LUT cascade) and coefficients table: by synchronous memory blocks; 2) Squaring unit: by logic elements; 3) Multiplier: by DSP units; and 4) Adder: by logic elements. Our synthesis system derives the appropriate bit-width for each component by automatic error analysis.

A. Size Reduction of Multiplier

Although modern FPGAs have dedicated multipliers, large multipliers are slow. In our architecture, the multiplier often has the longest delay time among all the units. Thus, to implement a fast NFG, reducing multiplier size is important. Since the size of multipliers depends on the number of bits for c_{2i}, c'_{1i}, and $x - q_i$, it is important to reduce the number of bits to represent these values.

First, we consider the case where the absolute values of c_{2i} and c'_{1i} are large. Our synthesis method uses a *scaling method* [9]. We represent c_{2i} and c'_{1i} as $c_{2i} = c_{2i} \times 2^{-l_{2i}} \times 2^{l_{2i}}$ and $c'_{1i} = c'_{1i} \times 2^{-l_{1i}} \times 2^{l_{1i}}$, respectively. That is, instead of the original values of c_{2i} and c'_{1i}, we store the values of $c_{2i} \times 2^{-l_{2i}}$, l_{2i}, $c'_{1i} \times 2^{-l_{1i}}$, and l_{1i} in the coefficients table. In this case, the products $c_{2i}(x - q_i)^2$ and $c'_{1i}(x - q_i)$ are computed using multipliers and shifters. The use of l_{2i} and l_{1i} reduces the number of bits to represent the values of $c_{2i} \times 2^{-l_{2i}}$ and $c'_{1i} \times 2^{-l_{1i}}$, but increases the rounding errors. Our synthesis method finds optimum values of l_{2i} and l_{1i} for each segment such that an acceptable error is achieved. When l_{2i} and l_{1i} are 0 for all the segments, no shifter is implemented, that is, $c_{2i}(x - q_i)^2$ and $c'_{1i}(x - q_i)$ are directly implemented with multipliers.

Next, we consider the value of $x - q_i$. The number of bits for $x - q_i$ influences the sizes of the squaring unit and multipliers. Thus, reducing the value of $x - q_i$ reduces the sizes of the squaring unit and multipliers, and also the error. From (5), we can choose any value for q_i. To reduce the value of $x - q_i$, for a segment $[s_i, e_i]$, we set $q_i = (s_i + e_i)/2$. Then, we have $|x - q_i| \leq (e_i - s_i)/2$. Thus, reducing the segment width $e_i - s_i$ reduces the value for $x - q_i$. However, this also increases the number of segments, and results in increased memory size. The rest of this section shows a reduction method of segment width without increasing the memory size.

The coefficients table in Fig. 3 has 2^k words, where $k = \lceil \log_2 t \rceil$ and t is the number of segments. Therefore, we can increase the number of segments up to $t = 2^k$ without increasing the memory size. From Theorem 4.1, the size of LUT cascade also depends on the value of k. However, increasing the number of segments to $t = 2^k$ seldom increases the size of the LUT cascade. We reduce the size of segments by dividing the largest segment into two equal sized segments up to $t = 2^k$. This method reduces both the number of bits for $x - q_i$ and the error without increasing the memory size.

B. Pipeline Processing

To implement a high-throughput NFG in an FPGA, our synthesis system inserts pipeline registers between all units in the architecture. Since all units operate in parallel, and each unit has a short delay time, our NFGs achieves high throughput. Table I shows the units and the number of pipeline stages for them. Our NFGs have $n\_cas + (5 \text{ or } 6)$ pipeline stages, where n_cas is the number of LUTs for the LUT cascade.

TABLE II
NUMBER OF SEGMENTS FOR VARIOUS APPROXIMATION METHODS.

Function	Domain	AAE $= 2^{-17}$				AAE $= 2^{-25}$			
$f(x)$		Linear	2nd-Chebyshev		Time	Linear	2nd-Chebyshev		Time
		Non	Uniform	Non	[msec]	Non	Uniform	Non	[msec]
2^x	[0, 1]	128	9	7	0.1	2048	65	44	70
$1/x$	[1, 2)	124	16	11	0.1	1982	128	64	60
$\sqrt{x}$	[1/32, 2)	193	252	24	10	3082	2016	138	150
$1/\sqrt{x}$	[1, 2)	46	16	8	0.1	1024	128	46	50
$\log_2(x)$	[1, 2)	128	16	10	10	2048	128	56	70
$\ln(x)$	[1, 2)	89	16	9	10	1437	128	50	50
$\sin(\pi x)$	[0, 1/2)	127	17	12	10	2027	129	74	90
$\cos(\pi x)$	[0, 1/2)	127	17	12	10	2027	129	74	90
$\tan(\pi x)$	[0, 1/4)	112	33	12	10	1787	129	73	110
$\sqrt{-\ln(x)}$	[1/32, 1)	354	31744	52	70	5933	8126464	331	720
$\tan^2(\pi x)+1$	[0, 1/4)	256	33	17	20	4096	257	101	170
Entropy	[1/256, 255/256]	520	509	40	30	8320	4065	234	300
Sigmoid	[0, 1]	127	33	13	20	2020	129	76	160
Gaussian	[0, 1/2]	32	5	4	0.1	512	33	18	30
Average		170	2337	17	20	2739	580995	99	100

AAE: Acceptable Approximation Error. Time: CPU time for our non-uniform segmentation algorithm.
Linear: Linear approximation. 2nd-Chebyshev: 2nd-order Chebyshev approximation.
Uniform: Uniform segmentation. Non: Non-uniform segmentation.
Experiment environment: CPU: Pentium4 Xeon 2.8GHz Memory: 4GB
OS: Redhat (Linux 7.3) C compiler: gcc -O2

TABLE I
NUMBER OF PIPELINE STAGES FOR NFGS.

Name of units	Pipeline stages
1. Segment index encoder	*n_cas*
2. Coefficients table	1
3. Adder for $x+(-q_i)$	1
4. Squaring unit	1
5. Multipliers (parallel)	1
6. Shifter (optional)	0 or 1
7. Final adder	1
Total pipeline stages	$n\_cas + (5 \text{ or } 6)$

n_cas: Number of LUTs for LUT cascade.

VI. EXPERIMENTAL RESULTS

A. Number of Segments and Computation Time of Algorithm

Table II compares the number of segments for various approximation methods for the functions in [16]. In this table, *Entropy*, *Sigmoid*, and *Gaussian* are

$$\begin{aligned} Entropy &= -x\log_2 x-(1-x)\log_2(1-x), \\ Sigmoid &= \frac{1}{1+e^{-4x}}, \quad \text{and} \quad Gaussian = \frac{1}{\sqrt{2\pi}}e^{-\frac{x^2}{2}} \end{aligned}$$

In Table II, the columns "Linear Non" show the number of non-uniform segments for linear approximation in [17], and the columns "2nd-Chebyshev Uniform" and "2nd-Chebyshev Non" show the number of uniform segments and non-uniform segments for 2nd-order Chebyshev approximation, respectively. The columns "Time" show the CPU time for our non-uniform segmentation algorithm applied to functions, in milliseconds.

Table II shows that, for many functions, the 2nd-order Chebyshev approximations require many fewer segments than the linear approximation. However, for some functions, such as $\sqrt{-\ln(x)}$, the 2nd-order Chebyshev approximation based on uniform segmentation requires many more segments than the linear and 2nd-order Chebyshev approximations based on non-uniform segmentations. Many existing polynomial approximation methods are based on uniform segmentation. For trigonometric and exponential functions, approximation methods based on uniform segmentation require relatively few segments. However, for some kinds of functions such as $\sqrt{-\ln(x)}$, the uniform 2nd-order approximation method requires excessively many segments. On the other hand, our quadratic approximation based on non-uniform segmentation requires fewer segments for a wide range of functions. Also, Table II shows that the CPU time is strongly correlated to the number of segments. Smaller acceptable approximation error (AAE) requires more segments and longer computation time. However, Table II shows that, for all functions in the table, the CPU times are shorter than 1 second when the acceptable approximation error is 2^{-25}.

These results show that, for various functions, our segmentation algorithm partitions a domain into fewer non-uniform segments quickly, and it is useful for automatic synthesis.

B. Memory Sizes of Various NFGs

This section compares the memory sizes of our NFGs with three existing NFGs [17, 3, 4]. Table III compares NFGs using linear approximation shown in [17]. This linear approximation is based on non-uniform segmentation. In Table III, the columns "R" show the following values:

$$R = \frac{\text{memory size of quadratic approximation}}{\text{memory size of linear approximation}} \times 100.$$

Table III shows that NFGs using quadratic approximation require much smaller memory than ones using linear approximation. Especially, 24-bit precision NFGs using quadratic approximation can be implemented with only 4% of the memory size needed for a linear approximation. From the relation between precision and memory size shown in Table III, we can see that increasing the precision decreases the ratio of memory sizes in NFGs.

TABLE III
COMPARISON WITH LINEAR APPROXIMATION BASED ON NON-UNIFORM SEGMENTATION.

Function $f(x)$	16-bit precision			24-bit precision		
	Memory [bits]		R	Memory [bits]		R
	Linear	Quad.	[%]	Linear	Quad.	[%]
2^x	20992	1112	5	696320	19072	3
$1/x$	21248	2432	11	700416	19136	3
$\sqrt{x}$	43776	5536	13	1425408	86784	6
$1/\sqrt{x}$	10176	1104	11	343040	19008	6
$\log_2(x)$	20864	2464	12	694272	19072	3
$\ln(x)$	20096	2448	12	700416	19136	3
$\sin(\pi x)$	19456	2336	12	661504	38656	6
$\cos(\pi x)$	19584	2336	12	663552	38784	6
$\tan(\pi x)$	19712	2304	12	667648	38272	6
$\sqrt{-\ln(x)}$	74240	11264	15	2662400	173056	7
$\tan^2(\pi x)+1$	37632	4960	13	1290240	39040	3
Entropy	106496	10688	10	3768320	83968	2
Sigmoid	21120	2432	12	702464	40320	6
Gaussian	4416	444	10	156672	8384	5
Average	31415	3704	11	1080905	45906	4

Memory: Memory size. Linear: Linear approximation [17].
Quad.: 2nd-order Chebyshev approximation. R: Ratio.

TABLE IV
COMPARISON WITH 5TH-ORDER APPROXIMATION BASED ON UNIFORM SEGMENTATION.

Func. $f(x)$	Domain	Acc.	Memory size [bits]		Ratio [%]
			5th-order (Uniform)	Quad. (Non)	
$\sin(\pi x)$	[0, 1/4]	2^{-23}	70528	18048	26
$\exp(x)$	[0, 1]	2^{-24}	82432	43136	52
2^x-1	[0, 1]	2^{-24}	89600	19968	22

Acc.: Accuracy.
5th-order: 5th-order approximation [3].
Quad.: 2nd-order Chebyshev approximation.

Table IV and Table V compare our NFGs with NFGs using 5th-order Taylor expansion [3] and NFGs using 2nd-order minimax approximation by the Remez algorithm [4], respectively. Both approximations in [3, 4] are based on uniform segmentation. Thus, their NFGs require no segment index encoder. On the other hand, since our approximation is based on non-uniform segmentation, the memory size is obtained by the sum of the coefficients table and the segment index encoder. As shown in [17] and Table II, for trigonometric and exponential functions, the difference of the number of uniform segments and non-uniform segments is not so large under the same approximation polynomial. For such functions, NFGs based on uniform segmentation (needing no segment index encoder) often require smaller memory than non-uniform segmentations. Although our NFGs require the segment index encoder and use approximation polynomials with larger approximation error than approximation polynomials in [3, 4], our NFGs for such functions are implemented with only 22% to 52% of the memory sizes of NFGs in [3], and with memory size comparable to [4]. In [3, 4], memory sizes of NFGs for $\sqrt{x}$ and $\sqrt{-\ln(x)}$ are unavailable. However, from Table II, we can see that the memory size of their NFGs for $\sqrt{x}$ and $\sqrt{-\ln(x)}$ is excessively large. On the other hand, our NFGs can realize a wide range of functions with small memory size.

TABLE V
COMPARISON WITH QUADRATIC APPROXIMATION BASED ON UNIFORM SEGMENTATION.

Func. $f(x)$	Domain	Acc.	Memory size [bits]		Ratio [%]
			Minimax (Uniform)	Cheb. (Non)	
$\sin(\pi x/4)$	[0, 1)	2^{-24}	16288	19200	118
2^x-1	[0, 1)	2^{-16}	2208	2512	114

Minimax: 2nd-order minimax approximation [4].
Cheb.: 2nd-order Chebyshev approximation.

C. FPGA Implementation Results

Table VI compares the FPGA implementation results of our NFGs with NFGs using linear approximation [17].

Since the architecture of linear NFG is simpler than quadratic NFG, linear NFGs are faster, and require fewer logic elements and DSP units than quadratic NFGs. However, linear approximation requires more segments and larger memory than quadratic approximation, as shown in Table II and Table III. Table VI shows that 24-bit precision linear NFGs cannot realize any function except *Gaussian* with the FPGA (the smallest device in the Stratix family) due to the excessive memory size although many logic elements and DSP units are unused. The most crucial issue in the FPGA implementation is the constraints on these hardware resources. For 24-bit precision, the linear approximation requires a larger FPGA due to the excessive memory size. However, in the larger FPGA, more logic elements and DSP units are left unused and wasted. On the other hand, the quadratic NFGs can be implemented with a smaller FPGA since they require much less memory size than the linear NFGs and reasonable sizes of logic elements and DSP units. In fact, 24-bit precision quadratic NFGs can be implemented with lower cost and more compact FPGAs (Cyclone II).

VII. CONCLUSION AND COMMENTS

We have demonstrated an architecture and a synthesis method for programmable NFGs for trigonometric functions, logarithm functions, square root, reciprocal, etc. Our architecture can efficiently realize any non-uniform segmentation using a compact LUT cascade, and approximate many numerical functions by quadratic polynomials. Therefore, our architecture is suitable for automatic synthesis of fast and compact NFGs. Implementation results on an FPGA show that our synthesis method can approximate a wide range of functions with a small number of non-uniform segments, and generate NFGs with small memory size. For 24-bit precision, our NFGs can be implemented with only 4% of the memory size of NFGs based on the linear approximation with non-uniform segmentation, and with only 22% of the memory size of NFGs based on the 5th-order approximation with uniform segmentation. NFGs based on the linear approximation are faster than the quadratic ones, but for high-precision, they require a large FPGA due to the excessive memory size. On the other hand, our quadratic NFGs can be implemented with more compact and low-cost FPGA by using hardware resources on the FPGA efficiently.

ACKNOWLEDGMENTS

This research is partly supported by the Grant in Aid for Scientific Research of the Japan Society for the Promotion of

TABLE VI

FPGA IMPLEMENTATION OF NFGS FOR LINEAR AND QUADRATIC APPROXIMATIONS.

FPGA device:	Altera Stratix (EP1S10F484C5: 10570 logic elements, 48 DSP units)
Logic synthesis tool:	Altera QuartusII 5.0
Synthesis options:	speed optimization, timing requirement: 200MHz

Function	16-bit precision						24-bit precision					
$f(x)$	Logic elements		DSP units		Freq. [MHz]		Logic elements		DSP units		Freq. [MHz]	
	Linear	Quad.	Linear	Quad.	Linear	Quad.	Linear	Quad.	Linear	Quad.	Linear	Quad.
2^x	167	482	2	4	195	185	604	758	2	10	–	131
$1/x$	204	376	2	4	234	186	636	859	2	10	–	134
$\sqrt{x}$	270	496	2	4	237	179	1211	822	2	16	–	124
$1/\sqrt{x}$	186	475	2	4	237	186	402	753	2	10	–	131
$\log_2(x)$	163	381	2	4	194	186	597	757	2	10	–	131
$\ln(x)$	170	379	2	4	197	185	416	863	2	10	–	131
$\sin(\pi x)$	154	424	2	4	197	192	480	646	8	10	–	134
$\cos(\pi x)$	172	354	2	4	237	179	412	647	8	10	–	131
$\tan(\pi x)$	234	382	2	4	237	178	655	604	2	10	–	131
$\sqrt{-\ln(x)}$	304	623	2	10	215	135	854	942	8	16	–	130
$\tan^2(\pi x)+1$	132	282	2	4	194	215	991	720	2	10	–	135
Entropy	141	403	2	4	235	206	1370	914	2	16	–	128
Sigmoid	167	430	2	4	194	191	627	706	2	10	–	131
Gaussian	181	419	2	4	237	186	303	747	2	10	216	129
Average	189	422	2	4	217	185	683	767	3	11	–	131

Linear: Linear approximation [17]. Quad.: 2nd-order Chebyshev approximation. Freq.: Operating frequency.
–: NFGs cannot be mapped into the FPGA due to the excessive memory size.
Memory sizes are omitted in this table (see Table III).

Science (JSPS), funds from Ministry of Education, Culture, Sports, Science, and Technology (MEXT) via Kitakyushu innovative cluster project, and NSA Contract RM A-54.

REFERENCES

[1] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," *Proc. of the 1998 ACM/SIGDA Sixth Inter. Symp. on Field Programmable Gate Array (FPGA'98)*, pp. 191–200, Monterey, CA, Feb. 1998.

[2] J. Cao, B. W. Y. Wei, and J. Cheng, "High-performance architectures for elementary function generation," *Proc. of the 15th IEEE Symp. on Computer Arithmetic (ARITH'01)*, Vail, Co, pp. 136–144, June 2001.

[3] D. Defour, F. de Dinechin, and J.-M. Muller, "A new scheme for table-based evaluation of functions," *36th Asilomar Conference on Signals, Systems, and Computers,*, Pacific Grove, California, pp. 1608–1613, Nov. 2002.

[4] J. Detrey and F. de Dinechin, "Second order function approximation using a single multiplication on FPGAs," *Proc. Inter. Conf. on Field Programmable Logic and Applications (FPL'04)*, pp. 221–230, 2004.

[5] N. Doi, T. Horiyama, M. Nakanishi, and S. Kimura, "Minimization of fractional wordlength on fixed-point conversion for high-level synthesis," *Proc. of Asia and South Pacific Design Automation Conference (ASPDAC'04)*, pp. 80–85, 2004.

[6] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," *Proc. of the 12th IEEE Symp. on Computer Arithmetic (ARITH'95)*, Bath, England, pp. 10–16, July 1995.

[7] T. Ibaraki and M. Fukushima, *FORTRAN 77 Optimization Programming*, Iwanami, 1991 (in Japanese).

[8] Y. Iguchi, T. Sasao, and M. Matsuura, "Realization of multiple-output functions by reconfigurable cascades," *International Conference on Computer Design: VLSI in Computers and Processors (ICCD'01)*, Austin, TX, pp. 388–393, Sept. 23–26, 2001.

[9] D.-U. Lee, W. Luk, J. Villasenor, and P. Y.K. Cheung, "Non-uniform segmentation for hardware function evaluation," *Proc. Inter. Conf. on Field Programmable Logic and Applications*, pp. 796–807, Lisbon, Portugal, Sept. 2003.

[10] D.-U. Lee, W. Luk, J. Villasenor, and P. Y.K. Cheung, "A hardware Gaussian noise generator for channel code evaluation," *Proc. of the 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'03)*, Napa, CA, pp. 69–78, April 2003.

[11] J. H. Mathews, *Numerical Methods for Computer Science, Engineering and Methematics*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1987.

[12] J.-M. Muller, *Elementary Function: Algorithms and Implementation*, Birkhauser Boston, Inc., Secaucus, NJ, 1997.

[13] S. Nagayama and T. Sasao, "Compact representations of logic functions using heterogeneous MDDs," *IEICE Trans. on fundamentals*, Vol. E86-A, No. 12, pp. 3168–3175, Dec. 2003.

[14] S. Nagayama, T. Sasao, and J. T. Butler, "Error analysis for programmable numerical function generators based on quadratic approximation," http://www.lsi-cad.com/Error-QNFG/.

[15] T. Sasao, M. Matsuura, and Y. Iguchi, "A cascade realization of multiple-output function for reconfigurable hardware," *Inter. Workshop on Logic Synthesis (IWLS'01)*, Lake Tahoe, CA, pp. 225–230, June 12–15, 2001.

[16] T. Sasao, J. T. Butler, and M. D. Riedel, "Application of LUT cascades to numerical function generators," *Proc. the 12th workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI'04)*, Kanazawa, Japan, pp. 422–429, Oct. 2004.

[17] T. Sasao, S. Nagayama, and J. T. Butler, "Programmable numerical function generators: architectures and synthesis method," *Proc. Inter. Conf. on Field Programmable Logic and Applications (FPL'05)*, Tampare, Finland, pp. 118–123, Aug. 2005.

[18] Scilab 3.0, INRIA-ENPC, France, http://scilabsoft.inria.fr/

[19] M. J. Schulte and J. E. Stine, "Approximating elementary functions with symmetric bipartite tables," *IEEE Trans. on Comp.*, Vol. 48, No. 8, pp. 842–847, Aug. 1999.

[20] J. E. Stine and M. J. Schulte, "The symmetric table addition method for accurate function approximation," *Jour. of VLSI Signal Processing*, Vol. 21, No. 2, pp. 167–177, June 1999.

[21] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Comput.*, Vol. EC-820, No. 3, pp. 330–334, Sept. 1959.