IEICE TRANS. FUNDAMENTALS, VOL.E89–A, NO.12 DECEMBER 2006

PAPER Special Section on VLSI Design and CAD Algorithms

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

Shinobu NAGAYAMA[†a)], **Tsutomu SASAO**[††b)], ***Members***, ***and*** **Jon T. BUTLER**[†††c)], ***Nonmember***

SUMMARY This paper presents an architecture and a synthesis method for compact numerical function generators (NFGs) for trigonometric, logarithmic, square root, reciprocal, and combinations of these 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. Experimental results show that: 1) our NFGs require, on average, only 4% of the memory needed by NFGs based on the linear approximation with non-uniform segmentation; 2) our NFG for $2^x - 1$ requires only 22% of the memory needed by the NFG based on a 5th-order approximation with uniform segmentation; and 3) our NFGs achieve about 70% of the throughput of the existing table-based NFGs using only a few percent of the memory. Thus, our NFGs can be implemented with more compact FPGAs than needed for the existing NFGs. Our automatic synthesis system generates such compact NFGs quickly.

key words: *LUT cascades, 2nd-order Chebyshev approximation, non-uniform segmentation, NFGs, automatic synthesis, FPGA*

1. Introduction

Numerical functions $f(x)$, such as trigonometric, logarithmic, square root, reciprocal, and combinations of these functions, are extensively used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics, etc. To compute elementary functions, iterative algorithms, such as the CORDIC (COordinate Rotation DIgital Computer) algorithm [1], [23], have been often used. Although the CORDIC algorithm achieves accuracy with compact hardware, its computation time is proportional to the precision (i.e. the number of bits). For a function composed of elementary functions, the CORDIC algorithm is slower, since it computes each elementary function sequentially. It is too slow for numerically intensive applications. Implementation by a single lookup table for $f(x)$ is simple and very 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.

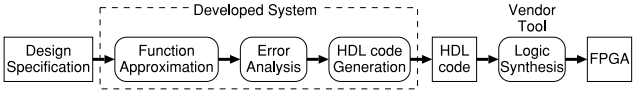


Fig. 1 Synthesis flow for NFGs.

To reduce the memory size, polynomial approximations have been used [9], [10], [21], [22]. These methods approximate the given numerical functions by piecewise polynomials, and realize the polynomials with hardware. Linear approximations offer fast and middle-precision evaluation of numerical functions. However, for high-precision, these methods also require excessive memory size. And, 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, and are realized by NFGs with small memory size. Our synthesis method is fully automated, so that fast and compact NFGs can be produced by non-experts. Figure 1 shows the synthesis flow for the NFG. It converts the Design Specification described by Scilab [19], a MATLAB-like software, into HDL code. The Design Specification consists of a function $f(x)$, a domain over x, and an accuracy. This system first partitions the domain into segments, and then approximates $f(x)$ by a quadratic function in 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.

2. Preliminaries

Definition 1: The ***binary fixed-point representation*** of a value r has the form

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

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

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

Manuscript received March 3, 2006.
Manuscript revised June 8, 2006.
Final manuscript received July 28, 2006.

[†]The author is with the Department of Computer Engineering, Hiroshima City University, Hiroshima-shi, 731–3194 Japan.
[††]The author is with the Department of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka-shi, 820–8502 Japan.
[†††]The author is with the Department of Electrical and Computer Engineering, Naval Postgraduate School, Monterey, CA 93943-5121 USA.

*This paper is an extension of [15].

a) E-mail: nagayama@ieee.org
b) E-mail: sasao@cse.kyutech.ac.jp
c) E-mail: jon_butler@msn.com

DOI: 10.1093/ietfec/e89–a.12.3510

Definition 2: ***Error*** is the absolute difference between the original value and the approximated value. ***Approximation error*** is the error caused by a function approximation. ***Rounding error*** is the error caused by a binary fixed-point representation. It is the result of truncation or rounding, whichever is applied. However, both operations yield an error that is called rounding error. ***Acceptable error*** is the maximum error that an NFG may assume. ***Acceptable approximation error*** is the maximum approximation error that a function approximation may assume.

For example, when a function is approximated by a piecewise linear function, approximation error occurs because the *approximating* function does not equal the *approximated* function everywhere. Typically, there is a non-zero error over most of the function values; indeed, this error is usually 0 only for a very few function values. Rounding error occurs because the binary representation of the function (2) takes on only specific values. For example, if the function value is $\sqrt{2}$, there is no representation in the form (2), where n_frac is finite, that is exactly $\sqrt{2}$.

Definition 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 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 an m-bit fractional part of the input, an m-bit fractional part of the output, and a 2^{-m} acceptable error.

Definition 5: ***Truncation*** is the process of removing lower order bits from a binary fixed-point number. If r is represented as $r = d_{n\_int-1}\ d_{n\_int-2}\ \ldots\ d_0.\ d_{-1}\ \ldots\ d_{-n\_frac}$ and is truncated to $r' = d_{n\_int-1}\ d_{n\_int-2}\ \ldots\ d_0.\ d_{-1}\ \ldots\ d_{-i}$, then the resulting rounding error is at most $2^{-i} - 2^{-n\_frac}$, where $i \le n\_frac$. Truncation of r never produces a value larger than r.

Definition 6: ***Rounding*** is truncation if $d_{-(i+1)} = 0$. If $d_{-(i+1)} = 1$, 2^{-i} is added to the result of truncation. That is, if r is represented as

$$r = d_{n\_int-1}\ d_{n\_int-2}\ \ldots\ d_0.\ d_{-1}\ \ldots\ d_{-n\_frac},$$

r is rounded to

$$r' = d_{n\_int-1}\ d_{n\_int-2}\ \ldots\ d_0.\ d_{-1}\ \ldots\ d_{-i} + d_{-(i+1)}2^{-i}.$$

The error caused by rounding is at most $2^{-(i+1)}$.

Truncation can cause a larger error than rounding. On the other hand, truncation requires less hardware. In our architecture, we use both truncation and rounding. This is discussed in Appendix.

3. Piecewise Quadratic Approximation

To approximate the numerical function $f(x)$ using quadratic functions, we first 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, we seek the fewest segments, since this reduces the memory size needed for storing the coefficients of the quadratic functions.

For piecewise polynomial approximations, in many cases, the domain is partitioned into uniform segments [2]–[4], [21], [22]. 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 uniform segmentation [9], [18]. However, a non-uniform segmentation requires an additional circuit that maps values of x to a segment number. Potentially, this is a complex circuit. To simplify the additional circuit, Lee et al. [9] have proposed a *special non-uniform segmentation* for the $\sqrt{-\ln(x)}$ function. Their method produces a simple circuit by restricting the segmentation points, and results in fewer segments as well as faster and more compact NFG than produced by uniform segmentation. However, it is not always optimum for the given function $f(x)$. For a fast and compact realization of *any non-uniform segmentations*, we use an LUT cascade proposed by Sasao et al. [17], [18] (see Sect. 4). By using the LUT cascade, we can use an optimum non-uniform segmentation for the given function $f(x)$. As far as we know, there is no other method that uses an optimum non-uniform segmentation.

3.1 Non-uniform Segmentation Algorithm

The number of non-uniform segments depends on the approximation polynomial. A more accurate approximation polynomial requires fewer segments. In this paper, we use the 2nd-order Chebyshev polynomials to approximate $f(x)$. We show that its coefficients are computed easily and quickly: it is suitable for fast automatic synthesis.

For a segment $[s, e]$ of $f(x)$, the maximum approximation error $\epsilon_2(s, e)$ of the 2nd-order Chebyshev approximation [11] is given by

$$\epsilon_2(s, e) = \frac{(e - s)^3}{192} \max_{s \le x \le e} |f^{(3)}(x)|, \tag{3}$$

where $f^{(3)}$ is the 3rd-order derivative of f. From (3), $\epsilon_2(s, e)$ 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 approximation error. Figure 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_a. Then, this algorithm approximates $f(x)$ with the acceptable approximation error ϵ_a, 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

Input:	Numerical function $f(x)$, Domain $[a, b]$ for x, Acceptable approximation error ϵ_a.
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 ($\geq s_i$) where $\epsilon_2(s_i, p) = \epsilon_a$.
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.

the value p where $\epsilon_2(s_i, p) = \epsilon_a$ is difficult. Thus, we obtain the maximum value p' satisfying $\epsilon_2(s_i, p') \leq \epsilon_a$. 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 specifying the bits of x from the MSB to the LSB such that $\epsilon_2(s_i, p') \leq \epsilon_a$. This requires a search with time complexity $O(n)$. In the computation of $\epsilon_2(s_i, p')$, the value of $\max_{s_i \leq x \leq p'} |f^{(3)}(x)|$ is computed by a nonlinear programming algorithm [7].

3.2 Computation of the 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 by $y = g_i(x) = c_{2i}x^2 + c_{1i}x + c_{0i}$, 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 yields the transformation

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

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

$$g_i(x) = c_{2i}(x - q_i)^2 + c'_{1i}(x - q_i) + c'_{0i}. \quad (4)$$

This transformation reduces the multiplier size.

4. Architecture for NFGs

Figure 3 shows the architecture that realizes (4). It has 7 units: the segment index encoder (which produces the segment index i for $[s_i, e_i]$ given the value x); the coefficients table (which stores $-q_i$, c_{2i}, c'_{1i}, and c'_{0i}); an adder (which produces $x + (-q_i)$); a squaring unit; two multipliers; and the output adder.

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. NFGs in which the most significant bits directly drive the address inputs of a coefficient memory need no segment index encoder. On the other hand, our NFGs based on non-uniform segmentation require this additional circuit. Although NFGs based on non-uniform segmentation have a smaller coefficients table than those based on uniform segmentation, the size of segment index encoder may have to be considered to provide a fair comparison. In [9], to simplify the segment index encoder, a special non-uniform segmentation is used for $\sqrt{-\ln(x)}$. However, it does not correspond to an optimum segmentation. To produce compact NFGs for a wide range of functions, it is important to guarantee that the size of the segment index encoder is reasonable for *any* non-uniform segmentation. Our synthesis system uses the *LUT cascade* [8], [16], [17] shown in Fig. 4(b) to realize *any* given (optimum) $seg\_func(x)$. In an LUT cascade, the interconnecting lines between adjacent LUTs are called *rails*. The size of an LUT cascade depends on the number of rails. Sasao et al. [17] have shown that the size of an LUT cascade depends on the number of segments. Specifically,

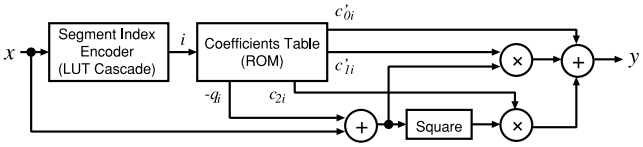
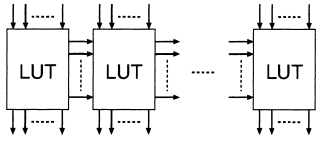


Fig. 3 Architecture for NFGs.

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

(a) Segment index function.



(b) LUT cascade.

Fig. 4 Segment index encoder.

Theorem 1: 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.

[18] shows that by using an LUT cascade, we can generate compact NFGs for a wide range of functions. In this paper, we significantly reduce memory sizes of the coefficients table and the LUT cascade by reducing the number of segments using the quadratic approximation. Our synthesis system uses heterogeneous MDDs (Multi-valued Decision Diagrams) [13], [14] to find compact LUT cascades. An LUT cascade passes data from the leftmost LUT to the rightmost LUT. Since each LUT in an LUT cascade operates independently and concurrently, we can easily obtain a pipeline structure by assigning each LUT to one pipeline stage. By using LUTs with the same size, we can easily achieve an efficient pipeline processing. To the best of our knowledge, this is the first method for realizing *any* $seg\_func(x)$.

4.1 Reduction of the Size of the Multiplier

To generate 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 $x - q_i$, we reduce the number of bits to represent these values.

To reduce the number of bits for c_{2i} and c'_{1i}, we use a *scaling method* [9]. We represent c_{2i} and c'_{1i} as $c_{2i} \times 2^{-l_{2i}} \times 2^{l_{2i}}$ and $c'_{1i} \times 2^{-l_{1i}} \times 2^{l_{1i}}$, respectively. And, we compute the

products $c_{2i}(x-q_i)^2$ and $c'_{1i}(x-q_i)$ using multipliers for the right-shifted multiplicand, $c_{2i}\times 2^{-l_{2i}}(x-q_i)^2$ and $c'_{1i}\times 2^{-l_{1i}}(x-q_i)$, and left shifters to restore the products. Applying right shifts reduces the number of bits for $c_{2i}\times 2^{-l_{2i}}$ and $c'_{1i}\times 2^{-l_{1i}}$ (i.e. the multiplier sizes) by rounding the l_{2i} LSBs of c_{2i} and l_{1i} of c'_{1i}, while increasing the rounding errors. In Appendix, we find the largest l_{2i} and l_{1i} for each segment that preserve the given acceptable error. When l_{2i} and l_{1i} are 0 for all the segments, we do not use scaling method.

Example 1: Consider a 24-bit precision NFG for $\sqrt{-\ln(x)}$. By using the scaling method, c_{2i}, l_{2i}, c'_{1i}, and l_{1i} have 13 bits, 5 bits, 20 bits, and 4 bits, respectively, and they produce an NFG with memory size: 173,056 bits, operating frequency: 130 MHz, and 16 DSP units. On the other hand, without the scaling method, c_{2i} and c'_{1i} have 37 bits and 28 bits, respectively, and they produce an NFG with larger memory size: 184,832 bits, much lower operating frequency: 71 MHz, and more DSP units: 20 units. (End of Example)

Next, we reduce 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 (4), 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 $|x-q_i|\le (e_i-s_i)/2$. 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. We show 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 1, the size of the LUT cascade also depends on the value of k. Thus, increasing the number of segments to $t=2^k$ rarely 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 reduces the rounding error as well (see Appendix).

5. Experimental Results

5.1 Number of Segments and Computation Time

Table 1 compares the number of segments for various approximation methods for the functions in [17]. In this table, *Entropy*, *Sigmoid*, and *Gaussian* are

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

In Table 1, the columns "Linear Uniform" and "Linear Non" show the number of uniform and non-uniform segments [18] for linear approximation, respectively. The columns "2nd-Chebyshev Uniform" and "2nd-Chebyshev Non" show the number of uniform and non-uniform segments for the 2nd-order Chebyshev approximation, respectively. The columns "Time" show the CPU time for our non-uniform segmentation algorithm applied to functions, in milliseconds.

Table 1 shows that, for many functions, the 2nd-order Chebyshev approximations require many fewer segments than the linear approximations. 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, the methods based on uniform segmentation require relatively few segments. However, for some kinds of functions such as $\sqrt{-\ln(x)}$, the uniform methods require excessively many segments, even if quadratic approximation is used. On the other hand, our quadratic approximation based on non-uniform segmentation requires many fewer

Table 1 Number of segments for various approximation methods.

Function $f(x)$	Domain	Acceptable approximation error: 2^{-17} (x has 15-bit accuracy)					Acceptable approximation error: 2^{-25} (x has 23-bit accuracy)				
		Linear		2nd-Chebyshev		Time	Linear		2nd-Chebyshev		Time
		Uniform	Non	Uniform	Non	[msec]	Uniform	Non	Uniform	Non	[msec]
2^x	[0, 1]	129	128	9	7	10	2,049	2,048	65	44	100
$1/x$	[1, 2)	128	124	16	11	10	2,048	1,982	128	64	90
$\sqrt{x}$	[1/32, 2)	2,016	193	252	24	10	32,256	3,082	2,016	138	130
$1/\sqrt{x}$	[1, 2)	128	46	16	8	10	2,048	1,024	128	46	70
$\log_2(x)$	[1, 2)	128	128	16	10	10	2,048	2,048	128	56	90
$\ln(x)$	[1, 2)	128	89	16	9	10	2,048	1,437	128	50	80
$\sin(\pi x)$	[0, 1/2)	257	127	17	12	10	4,097	2,027	129	74	70
$\cos(\pi x)$	[0, 1/2)	257	127	17	12	10	4,097	2,027	129	74	80
$\tan(\pi x)$	[0, 1/4)	257	112	33	12	10	4,097	1,787	129	73	140
$\sqrt{-\ln(x)}$	[1/32, 1)	31,744	354	31,744	52	90	8,126,464	5,933	8,126,464	331	840
$\tan^2(\pi x)+1$	[0, 1/4)	513	256	33	17	20	8,193	4,096	257	101	200
Entropy	[1/256, 255/256]	2,033	520	509	40	30	32,513	8,320	4,065	234	260
Sigmoid	[0, 1]	129	127	33	13	20	2,049	2,020	129	76	220
Gaussian	[0, 1/2]	33	32	5	4	10	513	512	33	18	30
Average		2,706	170	2,337	17	19	587,466	2,739	580,995	99	171

Linear: Linear approximation. 2nd-Chebyshev: 2nd-order Chebyshev approximation.
Uniform: Uniform segmentation. Non: Non-uniform segmentation.
Time: CPU time for our non-uniform segmentation algorithm conducted on the following environment.
System: Sun Blade 2500 (Silver), CPU: UltraSPARC-IIIi 1.6 GHz, Memory: 6 GB, OS: Solaris 9, and C compiler: gcc -O2.

Table 2 Comparison with linear approximation based on non-uniform segmentation.

Function $f(x)$	16-bit precision (15-bit accuracy)			24-bit precision (23-bit accuracy)		
	Memory [bits]		R	Memory [bits]		R
	Linear	Quad.	[%]	Linear	Quad.	[%]
2^x	20,992	1,112	5	696,320	19,072	3
$1/x$	21,248	2,432	11	700,416	19,136	3
$\sqrt{x}$	43,776	5,536	13	1,425,408	86,784	6
$1/\sqrt{x}$	10,176	1,104	11	343,040	19,008	6
$\log_2(x)$	20,864	2,464	12	694,272	19,072	3
$\ln(x)$	20,096	2,448	12	700,416	19,136	3
$\sin(\pi x)$	19,456	2,336	12	661,504	38,656	6
$\cos(\pi x)$	19,584	2,336	12	663,552	38,784	6
$\tan(\pi x)$	19,712	2,304	12	667,648	38,272	6
$\sqrt{-\ln(x)}$	74,240	11,264	15	2,662,400	173,056	7
$\tan^2(\pi x)+1$	37,632	4,960	13	1,290,240	39,040	3
Entropy	106,496	10,688	10	3,768,320	83,968	2
Sigmoid	21,120	2,432	12	702,464	40,320	6
Gaussian	4,416	444	10	156,672	8,384	5
Average	31,415	3,704	11	1,080,905	45,906	4

Memory: Memory size. Linear: Linear approximation [18].
Quad.: 2nd-order Chebyshev approximation. R: Ratio.

segments for a wide range of functions. Also, Table 1 shows that the CPU time to compute the segmentation strongly depends on the number of segments. Smaller acceptable approximation error requires more segments and longer computation time. However, Table 1 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 fast synthesis.

5.2 Memory Sizes of Various NFGs

This section compares the memory sizes of our NFGs with three existing NFGs [3], [4], [18]. Table 2 compares with NFGs based on linear approximation and *non-uniform segmentation* shown in [18]. In Table 2, the columns "R" show the following values:

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

Table 2 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, on average, only 4% of the memory size needed for a linear approximation. From the relation between precision and memory size shown in Table 2, we can see that increasing the precision decreases the ratio of memory sizes in NFGs.

Figure 5 shows a plot of the total memory size and the required precision for NFGs based on linear and quadratic approximations of $1/x$. Since memory size of quadratic NFG increases more slowly than that of linear NFG, as the precision increases, we conjecture that for precisions higher than 24-bit, our quadratic NFGs will require reasonable memory size. Unfortunately, we could not verify that because of the precision of our NFG synthesis tool. Table 3 and Table 4 compare our NFGs with NFGs using a 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 summing the memory sizes of the coefficients table and the segment index encoder. As shown in Table 1, 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 1, 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

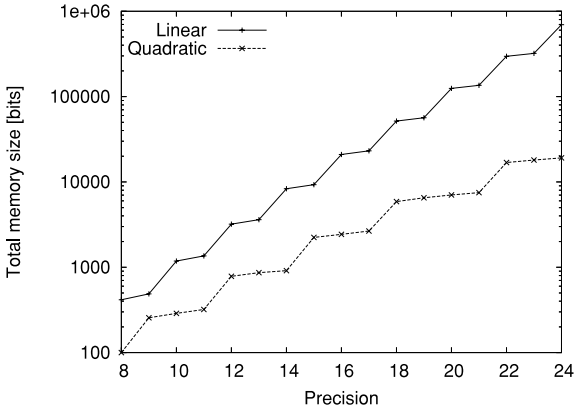


Fig. 5 Memory sizes of linear NFGs and quadratic NFGs for $1/x$.

Table 3 Comparison with 5th-order approximation based on uniform segmentation.

Func. $f(x)$	Domain	Accuracy	Memory size [bits]		Ratio [%]
			5th-order (Uniform)	Quad. (Non)	
$\sin(\pi x)$	[0, 1/4]	2^{-23}	70,528	18,048	26
$\exp(x)$	[0, 1]	2^{-24}	82,432	43,136	52
2^x-1	[0, 1]	2^{-24}	89,600	19,968	22

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

Table 4 Comparison with quadratic approximation based on uniform segmentation.

Func. $f(x)$	Domain	Accuracy	Memory size [bits]		Ratio [%]
			Minimax (Uniform)	Cheb. (Non)	
$\sin(\pi x/4)$	[0, 1)	2^{-24}	16,288	19,200	118
2^x-1	[0, 1)	2^{-16}	2,208	2,512	114

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

Table 5 FPGA implementation of NFGs for linear and quadratic approximations.

FPGA device:	Altera Stratix (EP1S10F484C5: 10,570 logic elements, 48 DSP units)
Logic synthesis tool:	Altera QuartusII 5.0 with speed optimization option and timing requirement of 200 MHz

Function $f(x)$	16-bit precision						24-bit precision					
	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	1,211	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	1,370	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 [18]. Quad.: 2nd-order Chebyshev approximation. Freq. : Operating frequency.
–: NFGs cannot be mapped into the FPGA due to the excessive memory size.

Table 6 Comparison with table-based method: STAM [22].

FPGA device:	Altera Stratix (EP1S25F780C5: 25,660 logic elements, 80 DSP units)
Logic synthesis tool:	Altera QuartusII 5.0 with speed optimization option and timing requirement of 200 MHz

Function $f(x)$	Domain	In prec.		Out prec.		Memory size [bits]			Operating frequency [MHz]		
		Int	Frac	Int	Frac	STAM	Quad.	Ratio [%]	STAM	Quad.	Ratio [%]
$1/x$	$[1, 2)$	1	23	1	23	651,264	19,136	3	184	131	71
$\sin(x)$	$[0, 1)$	0	24	0	24	491,520	40,832	8	205	131	64
2^x	$[0, 1)$	0	24	1	23	356,352	19,136	5	198	132	66

Quad.: our NFG. In prec.: Input precision. Out prec.: Output precision.
Int: Number of integer bits. Frac: Number of fractional bits.

size.

5.3 FPGA Implementation

This section compares the FPGA implementation results of our NFGs with two existing NFGs [18], [22]. Table 5 shows FPGA implementation results of our NFGs and NFGs based on linear approximation using non-uniform segmentation [18].

Since the architecture of a linear NFG [18] is simpler than that of a quadratic NFG, linear NFGs are faster, and require fewer logic elements and DSP units than quadratic NFGs. However, linear approximations require more segments and larger memory than quadratic approximations, as shown in Table 1 and Table 2. Table 5 shows that there are not enough resources in the smallest device in the Stratix family to achieve 24-bit precision using linear approximation for all functions, except *Gaussian*. This is 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 larger FPGAs, more logic elements and DSP units are left unused. On the other hand, a quadratic NFG can be implemented with a smaller FPGA, since it requires much smaller memory size than a linear NFG. In fact, we implemented 24-bit precision NFGs with lower cost and smaller FPGAs (Cyclone II), using quadratic approximation instead of linear approximation.

To show the performance of our NFGs, we compare with a well-known table-based NFG, STAM [22], that uses 1st-order Taylor expansion and uniform segmentation. Table 6 compares memory size and operating frequency. Since the STAM requires tables and a multiple-input adder, but requires no multiplier, it is faster. However, it requires excessive memory size and therefore larger FPGAs. On the other hand, our NFGs achieve about 60% to 70% of the operating frequency of the STAM with *much smaller memory size*.

6. Conclusion and Comments

We have demonstrated an architecture and a synthesis method for compact NFGs for trigonometric, logarithmic, square root, reciprocal, and combinations of these functions. Our architecture can efficiently realize any non-uniform segmentation using a compact LUT cascade, and can approximate many numerical functions by quadratic polynomials. Therefore, our architecture is suitable for automatic synthesis of fast and compact NFGs. Experimental results 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, on average, only 4% of the memory size of NFGs based on linear approximation and non-uniform segmentation, and with, on average, only 33% of the memory size of NFGs based on the 5th-order approximation and 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 achieve about 70% of the throughput of the table-based NFG using only a few percent of the memory, and can be implemented with a more compact and low-cost FPGA.

Our 24-bit precision NFGs can be used to compute a significand of an IEEE single-precision floating-point number. Since our NFGs are compact, we conjecture that our NFGs will be practical also for higher-precision than 24-bit. However, we could not verify the accuracy of NFGs with higher-precision than 24-bit due to the errors of NFG synthesis tool developed by C language. Thus, we have to develop a more accurate NFG synthesis tool and a verification tool in the future.

Acknowledgments

This research is partly supported by the Grant in Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS), funds from Ministry of Education, Culture, Sports, Science, and Technology (MEXT) via Kitakyushu innovative cluster project, NSA Contract RM A-54, and the Ministry of Education, Culture, Sports, Science, and Technology (MEXT), Grant-in-Aid for Young Scientists (B), 18700048, 2006. The comments of two referees were useful in improving this paper.

References

[1] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," Proc. 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. 15th IEEE Symp. on Computer Arithmetic (ARITH'01), pp.136–144, Vail, CO, 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, pp.1608–1613, Pacific Grove, California, 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, Leuven, Belgium, 2004.

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

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

[7] T. Ibaraki and M. Fukushima, FORTRAN 77 Optimization Programming, Iwanami, 1991.

[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), pp.388–393, Austin, TX, Sept. 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 Gaussian noise generator for hardware-based simulations," IEEE Trans. Comput., vol.53, no.12, pp.1523–1534, Dec. 2004.

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

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

[13] S. Nagayama and T. Sasao, "Compact representations of logic functions using heterogeneous MDDs," IEICE Trans. Fundamentals, vol.E86-A, no.12, pp.3168–3175, Dec. 2003.

[14] S. Nagayama and T. Sasao, "On the optimization of heterogeneous MDDs," IEEE Trans. Comput.-Aided Des. Integrated Circuits Syst., vol.24, no.11, pp.1645–1659, Nov. 2005.

[15] S. Nagayama, T. Sasao, and J.T. Butler, "Programmable numerical function generators based on quadratic approximation: Architecture and synthesis method," Proc. Asia and South Pacific Design Automation Conference (ASPDAC'06), pp.378–383, Yokohama, Japan, 2006.

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

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

[18] 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), pp.118–123, Tampere, Finland, Aug. 2005.

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

[20] M.J. Schulte and J.E. Stine, "Symmetric bipartite tables for accurate function approximation," 13th IEEE Symp. on Comput. Arith., vol.48, no.9, pp.175–183, Asilomar, CA, 1997.

[21] M.J. Schulte and J.E. Stine, "Approximating elementary functions with symmetric bipartite tables," IEEE Trans. Comput., vol.48, no.8, pp.842–847, Aug. 1999.

[22] J.E. Stine and M.J. Schulte, "The symmetric table addition method for accurate function approximation," J. VLSI Signal Processing, vol.21, no.2, pp.167–177, June 1999.

[23] J.E. Volder, "The CORDIC trigonometric computing technique," IRE Trans. Electronic Comput., vol.EC-820, no.3, pp.330–334, Sept. 1959.

Appendix: Error Analysis

We analyze the error for our NFGs and show a method to obtain the appropriate bit sizes for the units in our architecture. Our synthesis system applies the method in [5] to the NFG shown in Fig. 3.

A.1 Error of NFG

There are two kinds of errors: approximation error and rounding error. The approximation error is given by ϵ_a, as shown in Sect. 3. Thus, this section focuses on rounding error. We ignore the integer bits in this analysis because rounding errors are independent of integer bits. We assume there is at least one fractional bit.

Errors in Coefficients: The coefficients c_{2i}, c'_{1i}, and c'_{0i} for the quadratic approximation $g_i(x)$ are rounded to u_2-bit, u_1-bit, and u_0-bit accuracy, respectively. That is, due to rounding, the coefficients become

$$c_{2i} + \alpha_2, \quad c'_{1i} + \alpha_1, \quad \text{and} \quad c'_{0i} + \alpha_0,$$
$$(|\alpha_2| \le 2^{-(u_2+1)},\ |\alpha_1| \le 2^{-(u_1+1)},\ |\alpha_0| \le 2^{-(u_0+1)}),$$

where α_2, α_1, and α_0 are rounding errors of c_{2i}, c'_{1i}, and c'_{0i}, respectively. In this case, we need no hardware for rounding the coefficients because it is precomputed before storing in the coefficients table. Since rounding yields a more accurate result than the truncation, we choose rounding.

Error in Squaring Unit: When input x has m_{in} bits, our synthesis system ensures that q_i has also m_{in} bits. Therefore, the value of $(x - q_i)$ also has m_{in} bits, and has no rounding error. Although the value of $(x-q_i)^2$ has $2m_{in}$ bits, the value of $(x - q_i)^2$ is truncated to u_3 bits in order to reduce the size of succeeding multiplier, where $u_3 \le 2m_{in}$. We choose truncation because it does not require additional hardware, while rounding requires half adders at the output of squaring unit. Thus, the succeeding multiplier uses the value of

$$(x - q_i)^2 - \alpha_3 \qquad (0 \le \alpha_3 \le 2^{-u_3} - 2^{-2m_{in}}),$$

where α_3 is the rounding error for the truncation.

Errors in Multipliers: As shown in the two previous paragraphs, c_{2i} and c'_{1i} are stored as $c_{2i} + \alpha_2$ and $c'_{1i} + \alpha_1$ in the ROM, and $(x - q_i)^2$ is changed to $(x - q_i)^2 - \alpha_3$ by truncation. These values change the original products $c_{2i}(x - q_i)^2$ and $c'_{1i}(x - q_i)$ to

$$(c_{2i} + \alpha_2)\{(x - q_i)^2 - \alpha_3\} \text{ and} \tag{A·1}$$
$$(c'_{1i} + \alpha_1)(x - q_i). \tag{A·2}$$

The values of (A·1) and (A·2) have $(u_2 + u_3)$ bits and $(u_1 + m_{in})$ bits, respectively, and they are truncated to u_0 bits in order to match the addition with c'_{0i}. Note that $u_0 \le \min(u_2 + u_3, u_1 + m_{in})$. Following a method shown in [20], the $(u_0 + 1)$th-fractional bit $d_{-(u_0+1)}$ of (A·1) is set to 1 after the truncation. Thus, the adder uses the values of

$$(c_{2i} + \alpha_2)\{(x - q_i)^2 - \alpha_3\} - \alpha_4 \text{ and}$$
$$(c'_{1i} + \alpha_1)(x - q_i) - \alpha_5,$$

where α_4 and α_5 are the rounding errors for the truncations of (A·1) and (A·2): $0 \le \alpha_4 \le 2^{-(u_0+1)} - 2^{-(u_2+u_3)}$ and $0 \le \alpha_5 \le 2^{-u_0} - 2^{-(u_1+m_{in})}$.

Errors in Adder: From previous paragraphs, the original sum $c_{2i}(x - q_i)^2 + c'_{1i}(x - q_i) + c'_{0i}$ is changed to

$$(c_{2i} + \alpha_2)\{(x - q_i)^2 - \alpha_3\} + (c'_{1i} + \alpha_1)(x - q_i) + c'_{0i} + \alpha_0 - \alpha_4 - \alpha_5. \tag{A·3}$$

This value has u_0 bits, and is rounded to m_{out} bits (output accuracy given in the specification), where $m_{out} \le u_0$. Thus, the value of $g_i(x)$ is changed to

$$(c_{2i} + \alpha_2)\{(x - q_i)^2 - \alpha_3\} + (c'_{1i} + \alpha_1)(x - q_i) + c'_{0i} + \alpha_0 - \alpha_4 - \alpha_5 + \alpha_6,$$

where α_6 is the error for the rounding to m_{out} bits. Since $d_{-(u_0+1)}$ of (A·1) is set to 1, $d_{-(u_0+1)}$ of (A·3) is also 1, and we have $|\alpha_6| \le 2^{-(m_{out}+1)} - 2^{-(u_0+1)}$. Note that $d_{-(u_0+1)}$ is not implemented in hardware [20]. By expanding and rearranging this, we have

$$g_i(x) + \alpha_2\{(x - q_i)^2 - \alpha_3\} + \alpha_1(x - q_i) + \alpha_0 - c_{2i}\alpha_3 - \alpha_4 - \alpha_5 + \alpha_6. \tag{A·4}$$

This is an output value of the NFG including rounding error. (A·4) has the maximum rounding error when $\alpha_0 = -2^{-(u_0+1)}$, $\alpha_1 = -2^{-(u_1+1)}$, $\alpha_2 = -2^{-(u_2+1)}$, $\alpha_3 = 2^{-u_3} - 2^{-2m_{in}}$, $\alpha_4 = 2^{-(u_0+1)} - 2^{-(u_2+u_3)}$, $\alpha_5 = 2^{-u_0} - 2^{-(u_1+m_{in})}$, and $\alpha_6 = -(2^{-(m_{out}+1)} - 2^{-(u_0+1)})$, where we assume that the values of $(x-q_i)$ and c_{2i} are positive. Therefore, the maximum rounding error ϵ_r is

$$\epsilon_r = 2^{-(u_2+1)}(max\_seg^2 - 3 \cdot 2^{-u_3} + 2^{-2m_{in}}) + 2^{-(u_1+1)}max\_seg + 3 \cdot 2^{-(u_0+1)} + max\_c_2(2^{-u_3} - 2^{-2m_{in}}) - 2^{-(u_1+m_{in})} + 2^{-(m_{out}+1)},$$

where max_seg and max_c_2 are the maximum values of $|x - q_i|$ and $|c_{2i}|$, respectively.

A.2 Calculation of Bit Sizes for Units

The number of bits for the integer part is calculated as $\lceil \log_2(max\_value + 1) \rceil + 1$, where max_value is an integer, and denotes the maximum absolute value of the range.

On the other hand, the number of bits for the fractional part is calculated using the result of the error analysis. From the error analysis, an NFG with an acceptable error ϵ is achieved when $\epsilon_a + \epsilon_r < \epsilon$, where ϵ_a and ϵ_r are the maximum approximation error and the rounding error, respectively. To generate fast and compact NFGs, we find the minimum u_0, u_1, u_2, and u_3 that satisfy this relation using nonlinear programming [7] that minimizes $u_0 + u_1 + u_2 + u_3$ with the constraint $\epsilon_a + \epsilon_r < \epsilon$.

Calculation of Shift Bits l_{2i} and l_{1i}: From (A·1) and (A·2), the maximum errors in the multipliers ϵ_1 and ϵ_2 are

$$\epsilon_1 = 2^{-(u_2+1)}(max\_seg^2 - 2^{-u_3} + 2^{-2m_{in}}) + max\_c_2(2^{-u_3} - 2^{-2m_{in}}) \quad \text{and}$$
$$\epsilon_2 = 2^{-(u_1+1)}max\_seg,$$

respectively. Since max_seg and max_c_2 are the maximum values of $|x - q_i|$ and $|c_{2i}|$, respectively, there are positive integers l_{2i} and l_{1i} that satisfy the following relations for each segment:

$$2^{l_{2i}}2^{-(u_2+1)}\{|x - q_i|^2 - 2^{-u_3} + 2^{-2m_{in}}\} + |c_{2i}|(2^{-u_3} - 2^{-2m_{in}}) \le \epsilon_1 \quad \text{and}$$
$$2^{l_{1i}}2^{-(u_1+1)}|x - q_i| \le \epsilon_2.$$

We transform these into

$$2^{-(u_2-l_{2i}+1)}\{|x - q_i|^2 - 2^{-u_3} + 2^{-2m_{in}}\} + |c_{2i}|(2^{-u_3} - 2^{-2m_{in}}) \le \epsilon_1 \quad \text{and} \tag{A·5}$$
$$2^{-(u_1-l_{1i}+1)}|x - q_i| \le \epsilon_2. \tag{A·6}$$

From (A·5) and (A·6), for each segment, coefficients c_{2i} and

c'_{1i} can be rounded to $(u_2 - l_{2i})$ and $(u_1 - l_{1i})$ bits within an acceptable error, respectively. That is, l_{2i} and l_{1i} can be used for the scaling method in Sect. 4. From (A· 5) and (A· 6), we have

$$l_{2i} \le \log_2\left(\frac{nume.}{deno.}\right) \quad \text{and} \quad l_{1i} \le \log_2\left(\frac{max\_seg}{|x - q_i|}\right),$$

where

$$\begin{aligned} nume. &= 2^{-(u_2+1)}(max\_seg^2 - 2^{-u_3} + 2^{-2m_{in}}) \\ &\quad + (max\_c_2 - c_{2i})(2^{-u_3} - 2^{-2m_{in}}) \quad \text{and} \\ deno. &= 2^{-(u_0+1)}\{|x - q_i|^2 - 2^{-u_3} + 2^{-2m_{in}}\}. \end{aligned}$$

And, we choose

$$l_{2i} = \left\lfloor \log_2\left(\frac{nume.}{deno.}\right)\right\rfloor \text{ and } l_{1i} = \left\lfloor \log_2\left(\frac{max\_seg}{|x - q_i|}\right)\right\rfloor.$$

Shinobu Nagayama received the B.S. and M.E. degrees from the Meiji University, Kanagawa, Japan, in 2000 and 2002, respectively, and the Ph.D. degree in computer science from the Kyushu Institute of Technology, Iizuka, Japan, in 2004. He is now a research associate at the Hiroshima City University, Hiroshima, Japan. He received the Outstanding Contribution Paper Award from the IEEE Computer Society Technical Committee on Multiple-Valued Logic (MVL-TC) in 2005 for a paper presented at the International Symposium on Multiple-Valued Logic in 2004, and the Excellent Paper Award from the Information Processing Society of Japan (IPS) in 2006. His research interest includes numerical function generators, decision diagrams, software synthesis, and embedded systems.

Tsutomu Sasao received the B.E., M.E., and Ph.D. degrees in electronics engineering from Osaka University, Osaka, Japan, in 1972, 1974, and 1977, respectively. He has held faculty/research positions at Osaka University, Japan, the IBM T.J. Watson Research Center, Yorktown Heights, New York, and the Naval Postgraduate School, Monterey, California. He is now a Professor of the Department of Computer Science and Electronics at the Kyushu Institute of Technology, Iizuka, Japan. His research areas include logic design and switching theory, representations of logic functions, and multiple-valued logic. He has published more than nine books on logic design, including *Logic Synthesis and Optimization, Representation of Discrete Functions, Switching Theory for Logic Synthesis*, and *Logic Synthesis and Verification*, Kluwer Academic Publishers, 1993, 1996, 1999, and 2001, respectively. He has served as Program Chairman for the IEEE International Symposium on Multiple-Valued Logic (ISMVL) many times. Also, he was the Symposium Chairman of the 28th ISMVL held in Fukuoka, Japan, in 1998. He received the NIWA Memorial Award in 1979, Distinctive Contribution Awards from the IEEE Computer Society MVL-TC for papers presented at ISMVLs in 1986, 1996, 2003 and 2004, and Takeda Techno-Entrepreneurship Award in 2001. He has served as an Associate Editor of the *IEEE Transactions on Computers*. He is a fellow of the IEEE.

Jon T. Butler received the BEE and MEngr degrees from Rensselaer Polytechnic Institute, Troy, New York, in 1966 and 1967, respectively. He received the Ph.D. degree from The Ohio State University, Columbus, in 1973. Since 1987, he has been a professor at the Naval Postgraduate School, Monterey, California. From 1974 to 1987, he was at Northwestern University, Evanston, Illinois. During that time, he served two periods of leave at the Naval Postgraduate School, first as a National Research Council Senior Postdoctoral Associate (1980–1981) and second as the NAVALEX Chair Professor (1985–1987). He served one period of leave as a foreign visiting professor at the Kyushu Institute of Technology, Iizuka, Japan. His research interests include logic optimization and multiple-valued logic. He has served on the editorial boards of the *IEEE Transactions on Computers*, *Computer*, and IEEE Computer Society Press. He has served as the editor-in-chief of *Computer* and IEEE Computer Society Press. He received the Award of Excellence, the Outstanding Contributed Paper Award, and a Distinctive Contributed Paper Award for papers presented at the International Symposium on Multiple-Valued Logic. He received the Distinguished Service Award, two Meritorious Awards, and nine Certificates of Appreciation for service to the IEEE Computer Society. He is a fellow of the IEEE.