Programmable Numerical Function Generators for Two-Variable Functions

Shinobu Nagayama

Dept. of Computer and Network Engineering, Hiroshima City University, Hiroshima 731-3194, Japan

Jon T. Butler

Dept. of Electrical and Computer Engineering, Naval Postgraduate School, CA 93943-5121, USA

Tsutomu Sasao

Dept. of Computer Science and Electronics, Kyushu Institute of Technology, Iizuka 820-8502, Japan

Abstract

This paper proposes a design method and programmable architectures for numerical function generators (NFGs) of two-variable functions. To realize a two-variable function in hardware, we partition a given domain of the given function into segments, and approximate the function by a polynomial in each segment. This paper introduces two planar segmentation algorithms that efficiently partition a domain of a two-variable function. This paper also introduces two architectures that can realize a wide range of two-variable functions. Our architectures allow a systematic design of two-variable functions. FPGA implementation results show that, for a complicated function, our NFG achieves 58% *of memory size and 39% of delay time of a circuit designed using one-variable NFGs.*

1. Introduction

The ability to compute numerical functions at a high speed is important in many applications, including 3D computer graphics and digital signal processing [11]. However, most existing methods are intended only for one-variable functions [4, 9, 13–16], and only a few methods exist for multi-variable functions [5, 6, 17]. Since these papers [5, 6, 17] present hardwares dedicated to specific functions, different functions need different design methods. As far as we know, systematic design method for generic multi-variable functions has never been presented.

A straightforward design method for arbitrary multi-variable function is to use a single memory in which the address is a combination of values of variables and the content of that address is the corresponding value of function. This method is fast, but requires a 2^{mn}-word memory to implement an m-variable function with n bits for each variable. Even for small m and n, this method is impractical because of large memory size.

To produce a practical implementation, multi-variable functions are often designed using combination of one-variable function generators, multipliers, and adders [5, 6]. This design method reduces the required memory size. However, depending on the function implemented, it can produce a slow implementation because of its complicated hardware architecture. Also, complicated hardware architecture makes error analysis harder. That is, guaranteeing output accuracy becomes harder.

This paper proposes a systematic design method for two-variable functions. Since our design method is based on a piecewise polynomial approximation, hardware architectures are simple even for complicated functions. To approximate a given function using piecewise polynomials, this paper introduces two planar segmentation algorithms that partition a given domain of two-variable function efficiently. This paper also introduces two programmable architectures that can realize a wide range of two-variable functions.

The rest of this paper is organized as follows: Section 2 introduces a number representation and the decision diagrams used in this paper. Section 3 presents two planar segmentation algorithms. Section 4 presents two architectures for two-variable functions. Section 5 evaluates performance of our segmentation algorithms and architectures for specific two-variable functions. And, Section 6 concludes the paper. Error analysis for our NFGs is omitted because it is the almost same as [12, 15].

2. Preliminaries

2.1. Number Representation and Errors

Definition 1 *A value X represented by the **binary fixed-point representation** is denoted by*

$$X = (x_{l-1}\ x_{l-2}\ \dots\ x_1\ x_0.\ x_{-1}\ x_{-2}\ \dots\ x_{-m}),$$

*where $x_i \in \{0,1\}$, l is the number of bits in the integer part, and m is the number of bits in the fractional part. Each bit x_i contributes $2^i x_i$ to the **value** of X except, x_{l-1}, which contributes $-2^{l-1}x_{l-1}$. That is, the fixed-point representation is in two's complement.*

Definition 2 ***Error*** *is the absolute difference between the exact value and the value produced by the hardware.* ***Acceptable error*** *is the maximum error that an NFG may assume; it is usually a specification to be satisfied by the hardware.* ***Approximation error*** *is the error caused by a function approximation.* ***Acceptable approximation error*** *is the maximum approximation error that a function approximation may assume.* ***Rounding error*** *is the error caused by a binary fixed-point representation.*

Definition 3 ***Accuracy*** *is the number of bits in the fractional part of a binary fixed-point representation.* ***m-bit accuracy*** *specifies that m bits are used to represent the fractional part of the number. When the maximum error is* 2^{-m}*, the accuracy is no greater than* 1 ***unit in the last place (ULP)*** *[11]. In this paper, an* ***m-bit accuracy NFG*** *is an NFG with an m-bit fractional part of the inputs, an m-bit fractional part of the output, and a* 1 *ULP error.*

2.2. Decision Diagrams

Definition 4 *A binary decision diagram (BDD) [2, 10] is a rooted directed acyclic graph (DAG) representing a logic function. The BDD is obtained by recursively applying the Shannon expansion* $f = \overline{x_i} f_0 + x_i f_1$ *to the logic function. It consists of two terminal nodes representing function values* 0 *and* 1 *respectively, and non-terminal nodes labeled by input variables. Each non-terminal node has two unweighted outgoing edges,* 0*-edge and* 1*-edge, that correspond to the values of the input variable. The terminal nodes have no outgoing edges. We consider only ordered BDDs, where the order of the variables is the same for every path from the root node to a terminal node. We consider only reduced BDDs, where identical subtrees are combined into a single tree.*

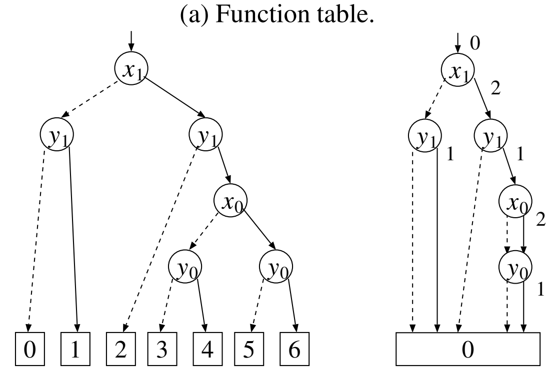
Definition 5 *A* ***multi-terminal BDD (MTBDD)*** *[3] is an extension of a BDD, that represents an integer-valued function:* $\{0,1\}^n \rightarrow Z$*, where Z is a finite set of integers. In the MTBDD, the terminal nodes are labeled by the values of Z.*

Definition 6 *An* ***edge-valued BDD (EVBDD)*** *[7, 8] is also an extension of a BDD, that represents an integer-valued function. The EVBDD is obtained by repeatedly applying the expansion* $f = \overline{x_i} f_0 + x_i (f_1' + \alpha)$ *to the integer-valued function, where* $f_1 = f_1' + \alpha$*, and* α *is the constant term of* f_1*. In the EVBDD, each* 1*-edge has an integer weight* α *and all* 0*-edges have weight* 0*. There is only one terminal node; it is labeled* 0*. The incoming edge into the root node can have a non-zero weight. For example, a non-zero weight* α *on the incoming edge of the root node adds* α *to all sums associated with the EVBDD. Indeed, it occurs when the EVBDD is a sub-EVBDD to a larger EVBDD.*

Example 1 *Fig. 1(b) and (c) show an MTBDD and an EVBDD for the integer-valued function* f *defined by Fig. 1(a). In Fig. 1(b) and (c), dashed lines and solid lines denote* 0*-edges and* 1*-edges, respectively. Note that the EVBDD has weighted* 1*-edges. In the MTBDD, terminal nodes represent function values. Thus, to evaluate the function, we traverse the MTBDD from the root node to a terminal node according to the input values, and obtain the function value (an integer) from the terminal node. On the other hand, in the EVBDD, we obtain the function value by summing the weights of the edges traversed from the root node to the terminal node.* *(End of Example)*

x_1	y_1	x_0	y_0	f	x_1	y_1	x_0	y_0	f
0	0	0	0	0	1	0	0	0	2
0	0	0	1	0	1	0	0	1	2
0	0	1	0	0	1	0	1	0	2
0	0	1	1	0	1	0	1	1	2
0	1	0	0	1	1	1	0	0	3
0	1	0	1	1	1	1	0	1	4
0	1	1	0	1	1	1	1	0	5
0	1	1	1	1	1	1	1	1	6

(a) Function table.



(b) MTBDD (c) EVBDD

Figure 1. MTBDD and EVBDD for an integer-valued function.

3. Piecewise Polynomial Approximation Based on Planar Segmentation

3.1. Planar Segmentation Problem

To approximate a given two-variable function by piecewise polynomials, we need to partition a given domain of the function into segments. The domain of a two-variable function consists of planar segments, and requires a planar segmentation algorithm. The memory size and speed of an NFG are strongly dependent on the efficiency of the segmentation algorithm. Thus, effective planar segmentation algorithms are important to design fast and compact NFGs. To produce an optimum segmentation, we consider the fol-

Input:	Numerical function $f(X,Y)$, domain $\{[X_b,X_e),[Y_b,Y_e)\}$ for X and Y, accuracy m_{in} of X and Y, polynomial order d, and acceptable approximation error ε_a.
*Requirement: X and Y are represented in the same number of bits.	
Output:	Segments $\{[X_b,P_0),[Y_b,Q_0)\},\{[X_b,P_0),[Q_0,Q_1)\}\ldots,\{[P_{r-1},X_e),[Q_{r-1},Y_e)\}$, and correction values $v_0,v_1,\ldots,v_{k-1}$.

Step:
1. For $\{[X_b,X_e),[Y_b,Y_e)\}$, compute an approximate polynomial $g_d(X,Y)$.
2. Compute the maximum positive error $max_{fg} = \max\{f(X,Y)-g_d(X,Y)\}$.
3. Compute the maximum negative error $min_{fg} = \min\{f(X,Y)-g_d(X,Y)\}$.
4. Compute approximation error $\varepsilon_d = (max_{fg}-min_{fg})/2$ and correction values $v = (max_{fg}+min_{fg})/2$.
5. If $\varepsilon_d < \varepsilon_a$ or $(X_e - X_b) \leq 2^{-m_{in}}$, then stop.
6. Else, partition $\{[X_b,X_e),[Y_b,Y_e)\}$ into four segments $\{[X_b,P),[Y_b,Q)\}$, $\{[X_b,P),[Q,Y_e)\}$, $\{[P,X_e),[Y_b,Q)\}$, and $\{[P,X_e),[Q,Y_e)\}$, where $P=(X_b+X_e)/2$ and $Q=(Y_b+Y_e)/2$.
7. Repeat Steps 1, 2, …, 6 for each new segment recursively, until the maximum approximation errors are smaller than ε_a in all segments.

Figure 2. Recursive planar segmentation algorithm.

lowing:

1. number of words in the *coefficients memory*, which is the number of segments, and
2. complexity of the *segment index encoder*, which maps values of X and Y to a segment number.

Fewer segments are preferred because the number of segments directly affects memory size of the NFG. The complexity of the segment index encoder is also important. Even if the number of segments is minimum, a large NFG is produced if the segment index encoder is very large. Especially, planar segmentations tend to require significantly more complex segment index encoders than linear segmentations. Thus, planar segmentation algorithms considering these two parameters are essential to the design of fast and compact NFGs. Also, the complexity of segmentation algorithms should be considered in order to reduce design time.

The next subsection presents two heuristic planar segmentation algorithms.

3.2. Planar Segmentation Algorithms

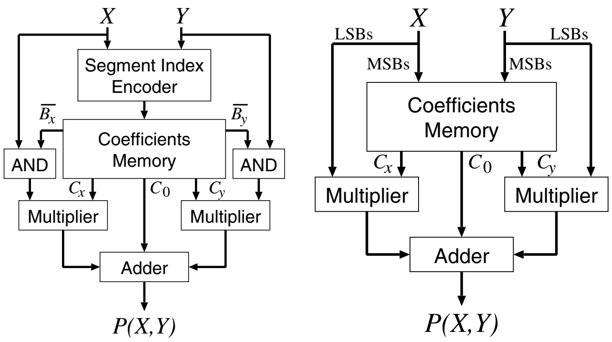
We first present a *recursive planar segmentation algorithm* to reduce the hardware complexity of both the coefficients memory (the number of segments) and the segment index encoder.

We provide a geometric explanation for piecewise planar (1st-order) polynomial approximation. Approximating a two-variable function is accomplished with parallelograms that project onto squares on the X-Y plane. First, a (large) single parallelogram is used to approximate the entire given function. It projects onto the X-Y plane as a square with corners at (X_b,Y_b), (X_b,Y_e), (X_e,Y_b), and (X_e,Y_e), where $X_e - X_b = Y_e - Y_b$. The parallelogram's orientation and altitude are chosen to minimize the maximum error. If this maximum error exceeds the given acceptable error, the following process is repeated. The projected square is divided into four squares each one fourth the area of the original square. This square is said to be *quadsected*. In each of the four sections, a parallelogram is determined that approximates the function with the smallest maximum error. If that error exceeds the given acceptable error, that square is quadsected, and the process repeated. The process stops when all square areas are approximated by a parallelogram to within the given acceptable error. It follows that, in areas where the function varies rapidly, small squares are used, and, in areas where the function is nearly planar, large squares are used.

Fig. 2 shows this algorithm. Note that this algorithm can apply to polynomial approximation with any degree. The inputs are a numerical function $f(X,Y)$, a domain $\{[X_b,X_e),[Y_b,Y_e)\}$ for X and Y, an accuracy m_{in} of X and Y, a polynomial order d, and an acceptable approximation error ε_a. Then, this algorithm produces segments by recursively partitioning a segment into four equal-sized square segments until achieving the acceptable approximation error ε_a in all segments. Note that this algorithm creates a segment of size $w_i \times w_i$, where $w_i = 2^{h_i} \times 2^{-m_{in}}$ and h_i is an integer. That is, all the segmentation points P_i and Q_i are restricted to values of which the least significant h_i bits are 0 (i.e., $P_i = (\ldots\ p_{-j+1}\ p_{-j}\ 00\ \ldots\ 0)$, where $j = m_{in} - h_i$). In Fig. 2, the approximating polynomial $g_d(X,Y)$ is obtained by a Taylor expansion of $f(X,Y)$ at the center (u,v) of the segment:

$$g_d(X,Y) = f(u,v) + \left(s\frac{\partial}{\partial X} + t\frac{\partial}{\partial Y}\right) f(u,v) + \left(s\frac{\partial}{\partial X} + t\frac{\partial}{\partial Y}\right)^2 \frac{f(u,v)}{2!} + \ldots + \left(s\frac{\partial}{\partial X} + t\frac{\partial}{\partial Y}\right)^d \frac{f(u,v)}{d!},$$

where $s = X-u, t = Y-v, u = (B_x+E_x)/2$, and $v = (B_y+E_y)/2$. To reduce the approximation error, the maximum positive error max_{fg} and the maximum negative error min_{fg} are equalized by a vertical shift of $g_d(X,Y)$ with correction value v. Thus, the approximation error is $(max_{fg} -$

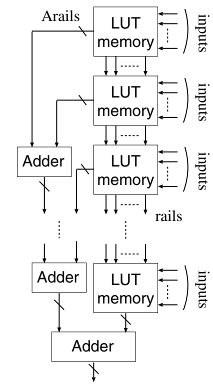


(a) Architecture for recursive segmentation. (b) Architecture for uniform segmentation.

Figure 3. Two architectures for two-variable NFGs based on planar approximation.

Segments	Index
$X_b \leq X < P_0$ $Y_b \leq Y < Q_0$	0
$X_b \leq X < P_0$ $Q_0 \leq X < Q_1$	1
$\vdots$	$\vdots$
$P_{r-1} \leq X < Y_e$ $Q_{r-1} \leq X < Y_e$	$k-1$

(a) Segment index function.



(b) LUT cascade and adders [13].

Figure 4. Segment index encoder.

$min_{fg})/2$, and the approximating polynomial used for the NFG is $g_d(X,Y)+v$.

Next, we present the *planar uniform segmentation algorithm*. Since the recursive planar segmentation algorithm produces *non-uniform* segmentation, the segment index encoder is needed to compute a segment number from values of X and Y. However, in a *uniform* segmentation where the number of segments is a power of 2, the segment index encoder is not necessary because a segment number is obtained by the most significant bits of X and Y (see Fig. 3(b)). This eliminates the delay of the segment index encoder, and produces fast NFGs. To produce uniform segmentation, we begin by finding the smallest square segment needed to achieve the acceptable approximation error using the recursive segmentation algorithm shown in Fig. 2. Then, we partition a given domain into square segments with the same size as the smallest segment.

4. Architectures for Two-Variable Numerical Function Generators

4.1. Architectures Based on Recursive and Uniform Segmentations

For each segment $\{[B_x,E_x),[B_y,E_y)\}$ produced by a planar segmentation algorithm, we compute the approximation to $f(X,Y)$ as a polynomial $P(X,Y)$ that is a Taylor expansion with a correction value. Expanding and rearranging the polynomial yields

$$\begin{aligned} P(X,Y) \quad = \quad & C_0 + C_x(X-B_x) + C_y(Y-B_y) \qquad (1) \\ & + C_{xy}(X-B_x)(Y-B_y) + C_{x2}(X-B_x)^2 \\ & + C_{y2}(Y-B_y)^2 + \ldots + C_{yd}(Y-B_y)^d. \end{aligned}$$

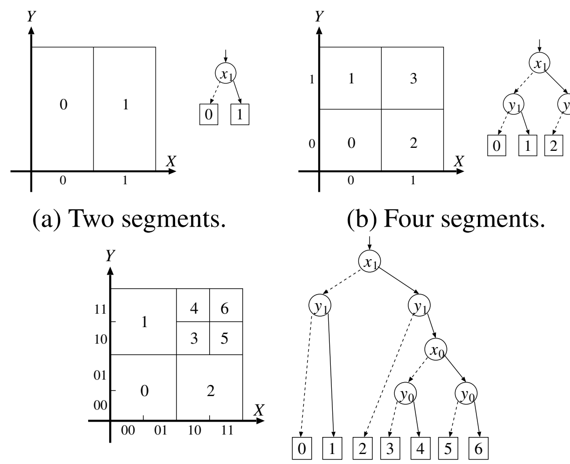
Fig. 3 shows two architectures for two-variable NFGs realizing (1) which use piecewise planar approximation (only the first three terms of (1). Expanding these architectures to a polynomial approximation with higher degree is straightforward. Fig. 3(a) and (b) show architectures based on recursive segmentation and uniform segmentations, respectively. The segment index encoder converts values of X and Y into a segment number. This, in turn, is applied as the address input of the Coefficients Memory. The coefficients are applied to adders and multipliers to form the polynomial value $P(X,Y)$. Note that Fig. 3(a) uses bitwise ANDs to compute $X-B_x$ and $Y-B_y$. In recursive segmentation, we can realize $X-B_x$ and $Y-B_y$ using AND gates driven on one side by $\overline{B_x}$ and $\overline{B_y}$, respectively [13].

Note that Fig. 3(b) has neither a segment index encoder nor bitwise ANDs. In uniform segmentation, the segment index encoder and bitwise ANDs are not necessary because a segment number, $X-B_x$, and $Y-B_y$ are obtained by the most significant bits and the least significant bits of X and Y, respectively. Since modern FPGAs have logic elements, synchronous memory blocks, and dedicated multipliers, these architectures are efficiently implemented by those hardware resources in an FPGA.

4.2. Architecture and Design Method for Segment Index Encoder

The segment index encoder realizes the segment index function: $\{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1,\ldots,k-1\}$ shown in Fig. 4(a), where X and Y have n bits, and k denotes the number of segments. We realize this function with the architecture shown in Fig. 4(b). In this architecture, the interconnecting lines between adjacent LUT memories determine the position in the EVBDD (labeled *rails*), and the outputs from each LUT memory to the adders tally the function value (labeled *Arails*). Consider the design of the LUT cascade and adders in Fig. 4(b), given the segmentation produced in Fig. 2.

We begin by representing the segment index function



(a) Two segments. (b) Four segments.

(c) Seven segments.

Figure 5. Relationship between recursive segmentation and MTBDDs.

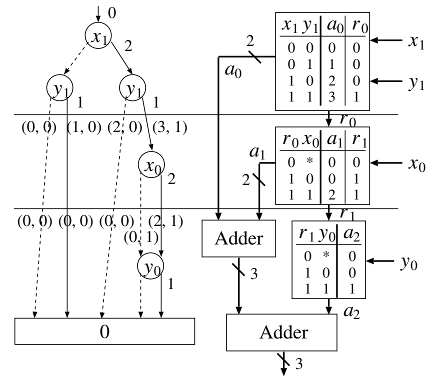


Figure 6. Decomposition of the EVBDD.

using an MTBDD. Fig. 5 illustrates the relationship between recursive segmentation and MTBDDs. Then, we convert the MTBDD into an EVBDD. By decomposing the EVBDD, as shown in Fig. 6, we obtain the architecture in Fig. 4(b). In Fig. 6, the column labeled as 'r_i' in the table of each LUT memory denotes the rails that represent subfunctions in the EVBDD. And, the column 'a_i' in Fig. 6 denotes the Arails that represent the sum of weights of edges. In the EVBDD, "(a_i, r_i)" assigned to edges that cut across the horizontal lines represents the sum of weights and subfunctions, respectively. For more detail on this architecture, see [13].

In this architecture, the size of LUT memories realizing the recursive segmentation depends on the number of segments. Specifically,

Theorem 1 *Let $seg\_func(X,Y)$ be a segment index function obtained by a recursive planar segmentation. The segment index function can be realized by the segment index encoder shown in Fig. 4(b) with at most $\lceil \log_2 k \rceil$ rails and $\lceil \log_2 k \rceil$ Arails, where k is the number of segments.*

Proof: See Appendix.

In our architectures, the coefficients memory and the LUT memories of the segment index encoder are implemented by embedded RAMs (e.g. M4Ks in Altera FPGAs). Thus, by changing the data for the coefficients memory and the LUT memories, a wide class of two-variable functions can be realized by a single architecture. Since just changing the RAM data can switch functions, we can switch functions without reprogramming the FPGA.

5. Experimental Results

5.1. Number of Segments and Computation Time for Algorithms

Table 1 shows the number of segments produced by the two segmentation algorithms presented in Section 3, and their computation time for various functions [1]. These segments are required to approximate two-variable functions by planar (1st-order) polynomials. In this table, *WaveRings*, *Gaussian*, and *Beta* are defined as:

$$WaveRings = \frac{\cos\left(\sqrt{X^2+Y^2}\right)}{\sqrt{X^2+Y^2+0.25}} \quad Gaussian = \frac{1}{Y\sqrt{2\pi}}e^{-\frac{X^2}{2Y^2}}$$

$$Beta = 2\int_0^{\frac{\pi}{2}} \sin^{2X-1}\theta \, \cos^{2Y-1}\theta \, d\theta = \frac{\Gamma(X)\Gamma(Y)}{\Gamma(X+Y)}$$

Table 1 shows that, for all functions except $\sin(\pi XY)$, the recursive segmentation algorithm produces many fewer segments than the uniform segmentation algorithm. Especially, for higher accuracy, the number of segments needed in recursive segmentation is only a few percent of the number of segments needed in uniform segmentation. For $\sin(\pi XY)$, the additional segments needed in uniform segmentation are not so large even for higher accuracy. This means that, for this function, the uniform segmentation method also produces an NFG with reasonable size. In addition, Table 1 shows that both algorithms produce segments with small CPU time. Such quick segmentation is useful to reduce design time for NFGs.

5.2. Memory Sizes Needed for Numerical Function Generators

Table 2 compares total memory sizes needed for NFGs based on the two planar approximation architectures shown in Fig. 3. Note that the NFGs based on recursive segmentation have two kinds of memories: coefficients memory and

Table 1. Number of segments for two segmentation methods based on planar approximation.

No.	Function $f(X,Y)$	Domain	X and Y have 8-bit accuracy (Acceptable approx. error: 2^{-10})					X and Y have 12-bit accuracy (Acceptable approx. error: 2^{-14})				
			No. of segments		R_s	Time [sec.]		No. of segments		R_s	Time [sec.]	
			Recur.	Uni.	[%]	Recur.	Uni.	Recur.	Uni.	[%]	Recur.	Uni.
f_0	$\sin(\pi X)\ln(Y)$	$0 \le X < 1, 0 < Y < 1$	4,696	65,280	7	0.19	0.02	244,807	16,773,120	1	8.9	7.2
f_1	$\sin(\pi X)\sqrt{Y}$	$0 \le X < 1, 0 < Y < 1$	1,393	16,384	9	0.07	0.61	38,773	16,773,120	0.2	1.7	6.7
f_2	$\sin(\pi XY)$	$0 \le X < 1, 0 \le Y < 1$	1,486	4,096	36	0.07	0.19	26,122	65,536	40	1.2	3.2
f_3	X^4Y^5	$0 \le X < 1, 0 \le Y < 1$	457	4,096	11	0.03	0.20	8,179	262,144	3	0.5	11.1
f_4	$1/\sqrt{X^2+Y^2}$	$0 < X < 1, 0 < Y < 1$	3,835	65,025	6	0.11	0.01	173,552	16,769,025	1	5.0	5.1
f_5	$XY/\sqrt{X^2+Y^2}$	$0 < X < 1, 0 < Y < 1$	376	4,096	9	0.01	0.11	6,523	1,048,576	0.6	0.2	22.5
f_6	*WaveRings*	$0 \le X \le \pi, 0 \le Y \le \pi$	1,619	10,201	16	0.08	0.39	28,377	646,416	4	1.3	18.9
f_7	*Gaussian*	$0 < X < 1, 0 < Y < 1$	3,182	65,025	5	0.12	0.02	141,113	16,769,025	0.8	5.5	7.1
f_8	$\sqrt{X^2+Y^2}$	$0 < X < 1, 0 < Y < 1$	355	4,096	9	0.01	0.12	6,160	1,048,576	0.6	0.2	24.6
f_9	$\sqrt[3]{X^3+Y^3}$	$0 < X < 1, 0 < Y < 1$	400	16,384	2	0.04	0.76	6,790	4,194,304	0.2	0.5	188.8
f_{10}	*Beta*	$1/8 \le X < 1, 1/8 \le Y < 1$	5,815	50,176	12	0.73	0.05	187,201	3,211,264	6	24.6	326.5

Recur.: Recursive segmentation. Uni.: Uniform segmentation. R_s: Recur. / Uni. $\times$ 100 (%).
Time: CPU time needed for segmentation algorithm.
Experiment environment
CPU: Intel Xeon 2.6GHz Memory : 1GB OS: Redhat Linux C compiler : gcc -O2

Table 2. Total memory sizes needed for NFGs based on two planar approximation architectures.

No.	8-bit accuracy NFGs			12-bit accuracy NFGs		
	Recursive	Uniform	R_m	Recursive	Uniform	R_m
f_0	260,052	783,360	33	16,683,510	285,143,040	6
f_1	59,511	360,448	17	2,030,356	201,277,440	1
f_2	69,352	110,592	63	1,313,684	2,293,760	57
f_3	25,392	102,400	25	516,230	8,126,464	6
f_4	226,403	1,040,400	22	13,054,030	402,456,600	3
f_5	18,120	90,112	20	369,189	27,262,976	1
f_6	100,030	346,834	29	1,886,924	23,917,392	8
f_7	186,980	910,350	21	11,345,482	368,918,550	3
f_8	16,882	94,208	18	316,128	28,311,552	1
f_9	21,792	278,528	8	405,576	88,080,384	0.5
f_{10}	291,735	602,112	48	11,814,069	122,028,032	10

R_m: Recursive / Uniform $\times$ 100 (%).

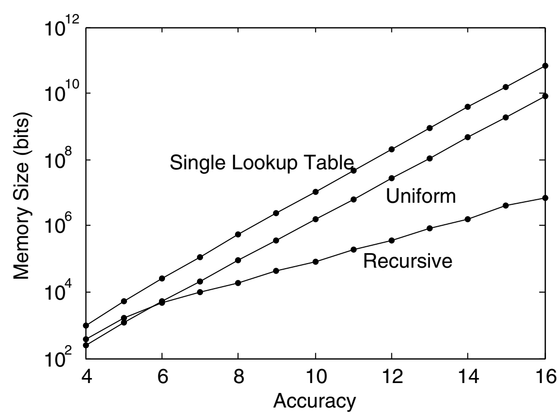


Figure 7. Memory size versus accuracy for $XY/\sqrt{X^2+Y^2}$.

LUT memory, and thus their memory size is the sum of the coefficients memory size and the LUT memory sizes.

Table 2 shows that, for all functions, NFGs based on recursive segmentation require smaller memory size than NFGs based on uniform segmentation, even though NFGs based on recursive segmentation have a segment index encoder. For example, for $XY/\sqrt{X^2+Y^2}$, the 12-bit accuracy NFG using recursive segmentation requires only 0.6% of memory required by uniform segmentation.

To understand the relation between memory size and accuracy, we designed NFGs for $XY/\sqrt{X^2+Y^2}$ with various accuracies. Fig. 7 plots memory sizes of the NFGs for 4 to 16-bit accuracies. There are three curves:

1. a single look-up table in which the values assigned to X and Y form an address and the contents of that address is $f(X,Y)$,
2. NFG with recursive non-uniform segmentation, and
3. NFG with uniform segmentation.

Interestingly, for this function, the memory size of the NFGs based on uniform segmentation increases in the same way as memory size of a single look-up table. On the other hand, the memory size of the NFGs based on recursive segmentation increases much more slowly than the other two. For 16-bit accuracy, the memory size of the NFG based on recursive segmentation is only 0.09% of that of the NFG based on uniform segmentation.

5.3. FPGA Implementation Results

We implemented 8-bit accuracy NFGs based on the two architectures using the Altera Stratix FPGA

Table 3. FPGA implementation of 8-bit accuracy NFGs based on two architectures.

FPGA device:	Altera Stratix (EP1S10F484C7)
Logic synthesis tool:	Synplify Pro Ver. 8.8

Function	Recursive segmentation					Uniform segmentation				
$f(X,Y)$	#LEs	#DSPs	Freq. [MHz]	#stages	Delay [nsec.]	#LEs	#DSPs	Freq. [MHz]	#stages	Delay [nsec.]
$\sin(\pi X)\ln(Y)$	347	4	149	9	60	–	0	–	1	–
$\sin(\pi X)\sqrt{Y}$	206	2	149	7	47	58	1	183	3	16
$\sin(\pi XY)$	280	2	169	7	41	62	2	183	3	16
X^4Y^5	280	2	169	7	41	57	2	183	3	16
$1/\sqrt{X^2+Y^2}$	552	4	169	9	53	–	0	–	1	–
$XY/\sqrt{X^2+Y^2}$	180	2	169	6	35	56	2	183	3	16
WaveRings	364	4	149	8	54	78	2	183	3	16
Gaussian	430	4	149	10	67	–	0	–	1	–
$\sqrt{X^2+Y^2}$	177	2	169	6	35	60	2	183	3	16
$\sqrt[3]{X^3+Y^3}$	189	2	169	6	35	56	0	343	3	9
Beta	439	4	169	9	53	–	0	–	1	–

–: NFGs cannot be mapped into the FPGA due to insufficient memory blocks.
#LEs: Number of logic elements. #DSPs: Number of 9-bit × 9-bit DSP units.
Freq. : Operating frequency. #stages : Number of pipeline stages.

(EP1S10F484C7). Table 3 compares the FPGA implementation results of the two architectures. In this table, the columns "Delay" show the total delay time of each NFG from the input to the output, in nanoseconds.

The NFGs based on uniform segmentation require fewer pipeline stages and have shorter delay than the recursive segmentation because they have no segment index encoder. However, for four functions, the NFGs based on uniform segmentation are not so easily implemented in an FPGA due to excessive memory size. Table 3 shows that they cannot be mapped into the FPGA due to insufficient memory blocks. Note that NFGs that have only one pipeline stage in Table 3 are realized with a single look-up table due to the excessively many segments. On the other hand, for all functions, the NFGs based on recursive segmentation achieve high operating frequency.

It is important to note that certain two-variable functions can be designed using 1. one-variable NFGs and 2. basic operations like addition and multiplication. For example, the first function in Table 1, $\sin(\pi X)\ln(Y)$, can be designed using two one-variable NFGs, one realizing $\sin(\pi X)$ and the other realizing $\ln(Y)$. The outputs are then multiplied together to realize the two-variable function. We are then interested in the complexity of this realization compared to the direct two-variable NFG design discussed earlier.

To understand the relative merits of using one versus two-variable NFGs, we implemented the following three functions from Table 1,

1. $\sin(\pi X)\ln(Y)$,
2. $XY/\sqrt{X^2+Y^2}$, and
3. *WaveRings*

using one-variable NFGs and basic operations. Each one-variable NFG was realized by the method shown in [13], which is based on linear approximation and non-uniform segment lengths. Table 4 shows the results.

Table 4. FPGA implementation of 8-bit accuracy NFGs designed using combination of one-variable NFGs.

FPGA device:	Altera Stratix (EP1S10F484C7)
Logic synthesis tool:	Synplify Pro Ver. 8.8

Function $f(X,Y)$	Memory [bits]	#LEs	#DSPs	Freq. [MHz]	#stages	Delay [nsec.]
$\sin(\pi X)\ln(Y)$	7,104	234	4	149	7	47
$XY/\sqrt{X^2+Y^2}$	31,104	381	13	133	12	90
WaveRings	15,232	410	10	149	11	74

Memory : Total memory size needed for two-variable functions.

Except for $\sin(\pi X)\ln(Y)$, the direct two-variable NFG implementation requires fewer logic elements (LEs) and DSPs than the one-variable implementation. Also, except for $\sin(\pi X)\ln(Y)$, the direct two-variable implementations have shorter delay. For $XY/\sqrt{X^2+Y^2}$ and *WaveRings*, the delays of the two-variable implementations are only 39% and 73% of those of the one-variable implementations, respectively. Especially, in the case of $XY/\sqrt{X^2+Y^2}$, both complexity and delay of the two-variable NFG are significantly less than the one-variable NFG implementation. For example, the X in the denominator, must be squared, added to Y^2, the reciprocal square root taken, and then multiplied by XY. This incurs a significant complexity and speed penalty.

From these results, we can see that by designing two-variable functions using one-variable NFGs, the required memory size can be reduced significantly. However, depending on functions, it can produce a slow implementation because of additional logic such as multipliers. Also, complicated hardware architecture using one-variable NFGs makes error analysis harder, and it is harder to guarantee output accuracy. This increases design time.

6. Concluding Remarks

We have proposed a design method and programmable architectures for numerical function generators of two-variable functions. To realize a two-variable function in hardware, we partition the given domain of the function into segments, and approximate the given function by a polynomial in each segment. In this paper, we presented two planar segmentation algorithms which partition a given domain of two-variable function efficiently. To the best of our knowledge, this is the first systematic design method based on piecewise polynomial approximation for two-variable functions. Experimental results show that for a complicated function, our automatically generated NFG achieves higher performance than manually designed NFG.

The algorithms and architectures presented in this paper can be easily extended to functions with three or more variables.

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, a contract with the National Security Agency, and the MEXT Grant-in-Aid for Young Scientists (B), 20700051, 2008.

References

[1] H. Anton, *Multivariable Calculus*, John Wiley & Sons, Inc., 1995.

[2] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677–691, Aug. 1986.

[3] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. Yang, "Spectral transforms for large Boolean functions with applications to technology mapping," *Proc. of 30th ACM/IEEE Design Automation Conference*, pp. 54–60, June 1993.

[4] J. Detrey and F. de Dinechin, "Table-based polynomials for fast hardware function evaluation," *16th IEEE Inter. Conf. on Application-Specific Systems, Architectures, and Processors (ASAP'05)*, pp. 328–333, 2005.

[5] R. Gutierrez and J. Valls, "Implementation on FPGA of a LUT based $atan(y/x)$ operator suitable for synchronization algorithms," *Proc. of the IEEE Conf. on Field Programmable Logic and Applications*, pp. 472–475, Aug. 2007.

[6] Z. Huang and M. D. Ercegovac, "FPGA implementation of pipelined on-line scheme for 3-D vector normalization," *Proc. of the 9th Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM'01)*, pp. 61–70, Apr. 2001.

[7] Y-T. Lai and S. Sastry, "Edge-valued binary decision diagrams for multi-level hierarchical verification," *Proc. of 29th ACM/IEEE Design Automation Conference*, pp. 608–613, 1992.

[8] Y-T. Lai, M. Pedram, and S. B. Vrudhula, "EVBDD-based algorithms for linear integer programming, spectral transformation and functional decomposition," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol. 13, No. 8, pp. 959–975, Aug. 1994.

[9] D.-U. Lee, W. Luk, J. Villasenor, and P. Y. K. Cheung, "Hierarchical segmentation schemes for function evaluation," *Proc. of the IEEE Conf. on Field-Programmable Technology*, Tokyo, Japan, pp. 92–99, Dec. 2003.

[10] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design: OBDD – Foundations and Applications*, Springer, 1998.

[11] J.-M. Muller, *Elementary Function: Algorithms and Implementation*, Birkhauser Boston, Inc., New York, NY, second edition, 2006.

[12] S. Nagayama, T. Sasao, and J. T. Butler, "Compact numerical function generators based on quadratic approximation: Architecture and synthesis method," *IEICE Trans. Fundamentals*, Vol. E89-A, No. 12, pp. 3510–3518, Dec. 2006.

[13] S. Nagayama, T. Sasao, and J. T. Butler, "Design method for numerical function generators using recursive segmentation and EVBDDs," *IEICE Trans. Fundamentals*, Vol. E90-A, No. 12, pp. 2752–2761, Dec. 2007.

[14] J.-A. Piñeiro, S. F. Oberman, J.-M. Muller, and J. D. Bruguera, "High-speed function approximation using a minimax quadratic interpolator," *IEEE Trans. on Comp.*, Vol. 54, No. 3, pp. 304–318, Mar. 2005.

[15] T. Sasao, S. Nagayama, and J. T. Butler, "Numerical function generators using LUT cascades," *IEEE Transactions on Computers*, Vol. 56, No. 6, pp. 826–838, Jun. 2007.

[16] 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.

[17] N. Takagi and S. Kuwahara, "A VLSI algorithm for computing the Euclidean norm of a 3D vector," *IEEE Transactions on Computers*, Vol. 49, No. 10, pp. 1074–1082, Oct. 2000.

Appendix

The proof of Theorem 1 is based on a theorem proven in [13]. Specifically, it was shown that

Theorem A *[13] Let $g(Z)$ be a k-valued monotone increasing function. The function $g(Z)$ can be realized by the segment index encoder shown in Fig. 4(b) with at most $\lceil \log_2 k \rceil$ rails and $\lceil \log_2 k \rceil$ Arails.*

Theorem 1 *Let $seg\_func(X,Y)$ be a segment index function obtained by a recursive planar segmentation. The segment index function can be realized by the segment index encoder shown in Fig. 4(b) with at most $\lceil \log_2 k \rceil$ rails and $\lceil \log_2 k \rceil$ Arails, where k is the number of segments.*

Proof: By forming a variable

$$Z = (x_{l-1}\ y_{l-1}\ x_{l-2}\ y_{l-2}\ \ldots\ x_{-m}\ y_{-m})$$

from X and Y, $seg\_func(X,Y)$ obtained by the recursive planar segmentation algorithm can be converted into a k-valued monotone increasing function $g(Z)$. Therefore, from Theorem A, we have this theorem. ∎