Design Method for Numerical Function Generators Based on Polynomial Approximation for FPGA Implementation

Shinobu Nagayama
Dept. of Computer Engineering,
Hiroshima City University,
Hiroshima 731-3194, Japan

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

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

Abstract

This paper focuses on numerical function generators (NFGs) based on k-th order polynomial approximations. We show that increasing the polynomial order k reduces significantly the NFG's memory size. However, larger k requires more logic elements and multipliers. To quantify this tradeoff, we introduce the FPGA utilization measure*, and then determine the optimum polynomial order k. Experimental results show that: 1) for low accuracies (up to* 17 *bits),* 1*st order polynomial approximations produce the most efficient implementations; and 2) for higher accuracies (*18 *to* 24 *bits),* 2*nd-order polynomial approximations produce the most efficient implementations.*

1. Introduction

With the introduction of FPGAs, it is possible to put, on one chip, large logic systems, including general purpose microprocessors and special system-on-a-chip designs. In spite of a large amount of available hardware, designers are often limited in their designs because a specific FPGA resource is scarce. That is, FPGAs consist of logic modules, multiplexers, adders, multipliers, and memory blocks. An application requiring many arithmetic modules, for example, may exhaust the adders and multipliers before exhausting memory modules. Therefore, the success of a design depends on achieving a balance on the use of various resources [17]. In this paper, we show a design of a numerical function generator (NFG) that adapts to the FPGA's resources; logic, arithmetic units, and memory.

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, 30], have been often used. Although the CORDIC algorithm achieves accuracy with compact hardware, its computation time is proportional to the number of bits used to represent the number. 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.

To reduce memory size, polynomial approximations have been used [3, 5, 6, 7, 12, 14, 22, 27, 28, 29]. These methods approximate the given numerical functions by piecewise polynomials, and realize the polynomials with hardware. For piecewise polynomial approximations, in many cases, the domain is partitioned into *uniform segments*. For elementary functions, such as $\sin(x)$ and e^x, by using higher-order polynomial approximations, the number of uniform segments can be reduced, and therefore the memory size can be reduced. However, for some numerical functions, such as $\sqrt{-\ln(x)}$, methods based on *uniform segmentation* yield large memory size for implementation on conventional FPGAs even if second-order polynomials are used [21]. On the other hand, since our NFG is based on *non-uniform segmentation*, for a wide range of functions, the memory size can be reduced by using second-order polynomials [21]. However, although second-order polynomial approximations reduce memory size, more multipliers and adders are required. To produce the most efficient FPGA implementation of NFGs, this paper introduces a measure, the *FPGA utilization measure*, and finds the polynomial order k that produces the FPGA implementations with the smallest *FPGA utilization measure*.

This paper focuses on the implementation of table lookup NFGs. Fig. 1 shows the synthesis flow of the de-

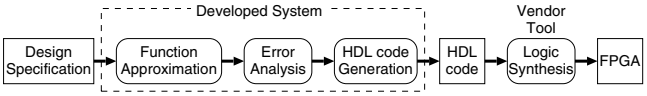


Fig. 1. Synthesis flow for NFGs.

sign process, which begins with a Design Specification described by Scilab [26], a MATLAB-like software application, and ends up with HDL code. The Design Specification consists of a function $f(x)$, a domain over x, an accuracy, and an order k of the approximation polynomial. This system first partitions the domain into segments, and then approximates $f(x)$ by a polynomial 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-supplied design software. The significance of the design flow is that it provides the context of the implementations shown here.

2. Preliminaries

Definition 1 *A **binary fixed-point representation** has the form $d_{l-1}\ d_{l-2}\ \ldots\ d_0.\ d_{-1}\ \ldots\ d_{-m}$, where $d_i \in \{0,1\}$ $(-m \le i \le l-1)$, l is the number of bits for the integer part, and m is the number of bits for the fractional part. This representation is two's complement. In this paper, we use $l = 1$.*

Definition 2 ***Error** is the absolute difference between the exact value and the value produced by the hardware. **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.*

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. In this paper, we assume that an **n-bit precision NFG** has an n-bit input and an acceptable error of 2^{-m}, where $m = n-1$.*

3. Piecewise Polynomial Approximation

To approximate the numerical function $f(x)$ using polynomial functions, we first partition the domain for x into segments. For each segment, we approximate $f(x)$ using a polynomial function $g(x) = c_k x^k + c_{k-1}x^{k-1} + \ldots + c_0$. In this case, we seek the fewest segments, since this reduces the memory size needed for storing the coefficients of the polynomial functions.

Input:	Numerical function $f(x)$, Domain $[a,b]$ for x, Acceptable approximation error ε_a, and Polynomial order k.
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 $\varepsilon_k(s_i,p) = \varepsilon_a$.
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.
6.	Else, let $s_i = p$, and go to step 2.

Fig. 2. Nonuniform segmentation algorithm for the domain.

Piecewise polynomial approximations have been applied to a domain that has been partitioned into *uniform segments* [3, 5, 6, 7, 22, 27, 28, 29]. Such methods are simple and fast, but for some kinds of numerical functions, too many segments are required, resulting in large memory. Further, for such functions, methods based on *uniform segmentation* cannot always reduce the memory size, even if the higher-order polynomials are used. For example, the reduction in the number of segments may not be sufficient to compensate for the increase in word width due to an increase in the number of stored coefficients needed for the higher-order polynomials.

For a given error, *non-uniform segmentation* of the domain uses fewer segments than uniform segmentation [12, 13, 25]. To reduce the memory size for a wide range of functions as the polynomial order increases, we use non-uniform segmentation.

3.1. Non-uniform Segmentation Algorithm

The number of non-uniform segments depends on the approximation polynomial. Specifically, fewer segments are required when the approximation polynomial is more accurate. In this paper, we use kth-order Chebyshev polynomials to approximate $f(x)$.

For a segment $[s,e]$ of $f(x)$, the maximum approximation error $\varepsilon_k(s,e)$ of the kth-order Chebyshev approximation [16] is given by

$$\varepsilon_k(s,e) = \frac{2(e-s)^{k+1}}{4^{k+1}(k+1)!} \max_{s \le x \le e} |f^{(k+1)}(x)|, \qquad (1)$$

where $f^{(k+1)}$ is the $(k+1)$th-order derivative of f. From (1), $\varepsilon_k(s,e)$ is a monotone increasing function of segment width $e-s$. From this property, it follows that a greedy algorithm in which each segment width is maximized for the given approximation error produces the optimum segmen-

tation. Fig. 2 shows the (nonuniform) segmentation algorithm. The inputs for this algorithm are a numerical function $f(x)$, a domain $[a,b]$ for x, an acceptable approximation error ε_a, and a polynomial order k. Then, this algorithm approximates $f(x)$ with 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 the value p, where $\varepsilon_k(s_i,p)=\varepsilon_a$, is difficult. We obtain the maximum value p' satisfying $\varepsilon_k(s_i,p')\le\varepsilon_a$ by scanning values of n-bit input x. However, it has time complexity $O(2^n)$. Therefore, we compute the maximum value p' by setting the bits of p' to 0 or 1 from the most significant to the least significant bits such that $\varepsilon_k(s_i,p')\le\varepsilon_a$, as in a binary search. This has time complexity $O(n)$. In the computation of $\varepsilon_k(s_i,p')$, the value of $\max_{s_i\le x\le p'}|f^{(k+1)}(x)|$ is computed by nonlinear programming [10].

3.2. Computation of Approximate Value

For each segment $[s_i,e_i]$, $f(x)$ is approximated by the corresponding polynomial function $g(x,i)$. That is, the approximated value y of $f(x)$ is computed as $y=g(x,i)=c_k(i)x^k+c_{k-1}(i)x^{k-1}+\ldots+c_0(i)$, where the coefficients $c_k(i)$, $c_{k-1}(i),\ldots,c_0(i)$ are derived from the kth-order Chebyshev approximation polynomial [16]. Substituting $x-q_i+q_i$ for x yields the transformation

$$\begin{aligned} g(x,i) &= c_k(i)(x-q_i)^k+c'_{k-1}(i)(x-q_i)^{k-1} \\ &\quad +\ldots+c'_0(i), \end{aligned} \tag{2}$$

where

$$c'_j(i)=\sum_{l=0}^{k-j}\binom{j+l}{j}c_{j+l}(i)q_i^l \quad (j=0,1,\ldots,k-1).$$

This transformation reduces the multiplier size (see Section 4.2). Instead of computing $g(x,i)$ in the form (2), we apply Horner's method [18] to derive (3) below:

$$\begin{aligned} g(x,i) &= ((c_k(i)(x-q_i)+c'_{k-1}(i))(x-q_i) \\ &\quad +\ldots)(x-q_i)+c'_0(i). \end{aligned} \tag{3}$$

This reduces the number of multipliers from approximately $\frac{k^2}{2}$ to k.

4. Architecture for NFGs

Fig. 3 shows the architecture realizing (3). It consists of the segment index encoder, the coefficients table, multipliers, and adders.

4.1. Segment Index Encoder

A *segment index encoder* converts an input x into a segment index i of corresponding segment $[s_i,e_i]$. It realizes the segment index function $seg\_func(x):\{0,1\}^n\rightarrow\{0,1,\ldots,t-1\}$ shown in Fig. 4 (a), where x has n bits, and t denotes the number of segments. NFGs based on uniform segmentation in which the most significant bits directly drive the address inputs of a coefficients table need no segment index encoder. On the other hand, NFGs based on non-uniform segmentation require this additional circuit. Potentially, this is a complex circuit.

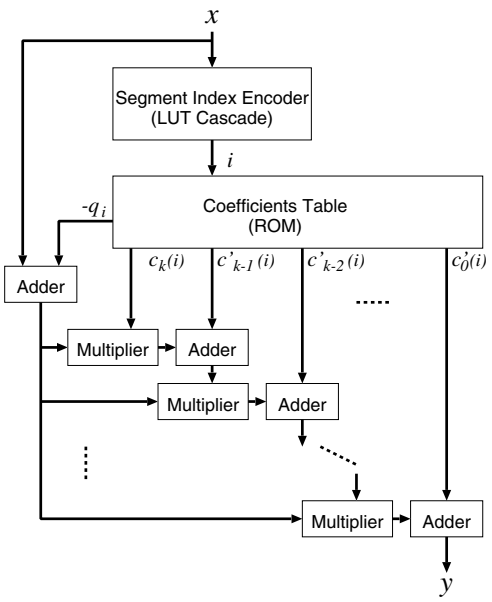
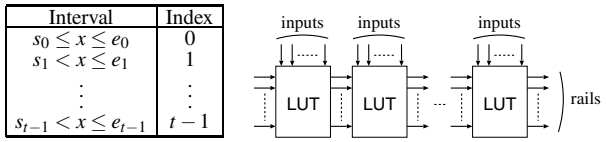


Fig. 3. Architecture for NFGs.



(a) Segment index function. (b) LUT cascade.

Fig. 4. Segment index encoder.

To simplify the segment index encoder, a *special non-uniform segmentation* [12, 13] has been proposed. This method produces a simple circuit by restricting the segmentation points, and results in fewer segments, as well as faster and more compact NFGs than produced by uniform segmentation. In this method, the user has to select a segmentation appropriate to the given function. Thus, it is difficult for non-experts to obtain optimum segmentation for the given function.

For a fast and compact realization of *any non-uniform segmentation*, we use an *LUT cascade* [11, 23] shown in Fig. 4 (b). By using an LUT cascade, for the given function, we can use the *optimum non-uniform segmentation* generated by the algorithm of Fig. 2. To obtain the LUT cascade, we consider $seg\_func(x)$ as a multiple-output logic function, and represent the logic function using a binary decision diagram (BDD) [2, 4]. By functional decompositions using the BDD, we obtain the LUT cascade. To produce

compact NFGs for a wide range of functions, it is important to guarantee that the size of LUT cascade is reasonable for *any* non-uniform segmentation. In [24], we have shown that the size of an LUT cascade depends on the number of segments, and by using an LUT cascade, we can generate compact NFGs for a wide range of functions. In this paper, we reduce the number of non-uniform segments using a high-order polynomial approximation, and thereby we significantly reduce the memory sizes of the coefficients table and the LUT cascade. Therefore, our NFGs can be implemented using remaining hardware resources in an FPGA. To the best of our knowledge, this paper presents the first NFG that uses kth-order approximating functions in optimum non-uniform segments, for $k > 2$.

4.2. Size Reduction of Multiplier

The number of bits representing $x - q_i$ determines the sizes of all the multipliers. Therefore, to reduce multiplier size, we reduce the number of bits representing $x - q_i$. Reducing the value of $x - q_i$ reduces not only the sizes of the multipliers, but also the error [8]. From (2), 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. We show a reduction method of segment width that does not increase memory size.

Assume that the coefficients table in Fig. 3 has 2^u words, where $u = \lceil \log_2 t \rceil$, and t is the number of segments. Therefore, we can increase the number of segments up to $t = 2^u$ without increasing the memory size. The size of an LUT cascade also depends on the value of u. However, increasing the number of segments to $t = 2^u$ rarely increases the size of the LUT cascade. We reduce the size of segments by dividing larger segments into two equal sized segments increasing t up to the next power of 2.

5. FPGA Utilization Measure

Modern FPGAs consist of various components, such as logic elements, memory blocks, and embedded multipliers. It is important to use these hardware resources efficiently. To generate the most efficient NFGs depending on the available hardware resources in an FPGA, we introduce the *FPGA utilization measure*.

Definition 4 *Given available hardware resources in an FPGA, the* ***FPGA utilization measure*** *U is the sum of utilizations for those hardware resources. In this paper, we assume that in an FPGA, there are four hardware resources: logic element (LE), embedded multiplier (DSP), and two types of RAM block (M4K and M512). That is,*

$$U = \left(\frac{R\_LE}{A\_LE} + \frac{R\_DSP}{A\_DSP} + \frac{R\_M4K}{A\_M4K} + \frac{R\_M512}{A\_M512} \right) \times 100\%,$$

where R_LE and A_LE denote the number of required LEs and available LEs, respectively. For DSP, M4K, and M512, we use a similar notation.

Using this measure, we find a polynomial order that produces the most efficient NFGs depending on the unused hardware resources in an FPGA. Specifically, we seek the smallest FPGA utilization measure across the various orders. Note that we can find a polynomial order that produces a feasible FPGA implementation by using a large penalty for the measure (e.g. $U = \infty$) when a required resource is larger than the available resource (e.g. $A\_LE < R\_LE$). However, our experimental results show that the smallest *FPGA utilization measure* results in a feasible and efficient FPGA implementation even if such penalty is not used.

By using NFGs with the smallest FPGA utilization measure, we can leave more hardware resources for other modules. This is useful in an incremental design of modules, such as occurs when a final design is the result of a sequence of specification changes.

Table 1. Numbers of uniform and non-uniform segments for $\sqrt{-\ln(x)}$ and $\arcsin(x)$.

Function $f(x)$	Domain $[a,b]$	k	Uniform segments	Non-uniform segments
$\sqrt{-\ln(x)}$	(0,1)	1	8,388,607	8,230 (0.0981%)
		2	8,388,607	698 (0.0083%)
		3	8,388,607	213 (0.0025%)
		4	8,388,607	111 (0.0013%)
		5	8,388,607	75 (0.0009%)
$\arcsin(x)$	[0,1)	1	8,388,608	3,067 (0.0366%)
		2	8,388,608	256 (0.0031%)
		3	8,388,608	81 (0.0010%)
		4	8,388,608	45 (0.0005%)
		5	4,194,304	31 (0.0007%)

6. Experimental Results

In this section, we find the optimum polynomial order for a given precision to approximate a function using the FPGA utilization measure discussed in the previous section.

6.1. Number of Segments and Memory Size

Table 1 compares the numbers of uniform and non-uniform segments for 24-bit precision NFGs of $\sqrt{-\ln(x)}$ $(0 < x < 1)$ and $\arcsin(x)$ $(0 \leq x < 1)$. From this table, we

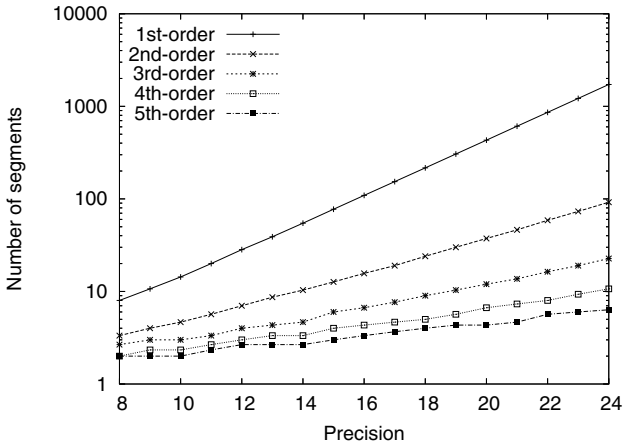


Fig. 5. Number of non-uniform segments versus precision.

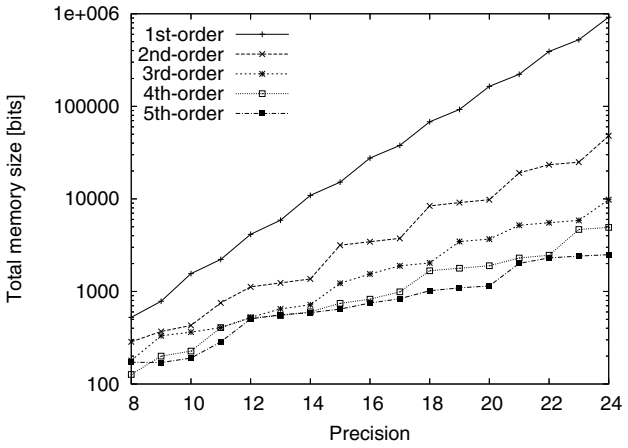


Fig. 6. Total memory size versus precision.

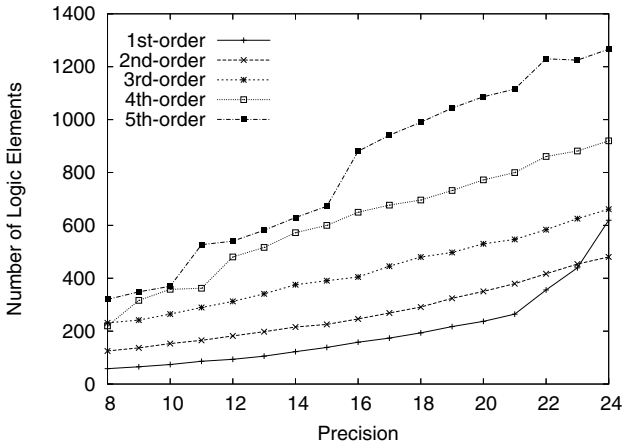


Fig. 7. Number of logic elements versus precision.

can see that the number of non-uniform segments significantly decreases as the polynomial order k increases, while the number of uniform segments does not always decrease. The number of segments determines the size of coefficients table. Many existing methods are based on uniform segmentation. Thus, for these functions, the existing methods cannot always reduce the memory size even if the polynomial order increases. On the other hand, our method can reduce the memory size by increasing the polynomial order for a wide range of functions.

In the following, we conduct experiments using three numerical functions: $\cos(\pi x)$ $(0 \leq x \leq 1/2)$, $\sqrt{x}$ $(1/32 \leq x < 2)$, and $1/x$ $(1 \leq x < 2)$. The experimental results shown in Fig. 5–13 are averages for these functions.

Fig. 5 shows the relation between the number of non-uniform segments and precision. Note the increase in the number of segments with precision, especially for 5th-order approximations. Further, there is a significant decrease in the number of segments as the order k of the polynomial increases. For example, for 24-bit precision, a 5th-order polynomial yields an approximation that uses only 0.37% of the segments needed in a 1st-order polynomial.

Fig. 6 shows the relation between total memory size and precision. That is, our NFGs require memory in the segment index encoder, as well as in the coefficients memory. Total memory size is the sum of these two. Fig. 6 shows that the total memory size increases exponentially with the precision. From Fig. 5 and Fig. 6, we can see that the total memory size strongly depends on the number of non-uniform segments. Thus, we can reduce the total memory size by increasing polynomial order. Especially, for high-precision, the increase of polynomial order reduces the memory size significantly. For 24-bit precision, the 5th-order polynomials require only 0.27% of total memory size needed for the 1st-order polynomials.

However, for low-precision, an increase of polynomial order does not always reduce the total memory size because, while it reduces the length of the coefficients table (i.e. the number of words of the coefficients table), it also increases the width of coefficients table (i.e. the bit-width of the coefficients table because more coefficients are needed in higher-order polynomials). In fact, for an 8-bit precision NFG for $\sqrt{x}$, the 5th-order polynomial requires more memory than the 4th-order polynomial (both polynomials require the same number of segments).

6.2. FPGA Resources

We implemented fully pipelined NFGs on the Altera Stratix EP1S80F1020C5 FPGA using the Quartus II ver. 5.0 development tool. We used the speed optimization option and set the required operating frequency to 200 MHz..

Fig. 7 shows the relation between the number of logic elements (LEs) and precision. This graph shows that the number of LEs increases approximately linearly with the precision. Further, increasing the polynomial order increases the

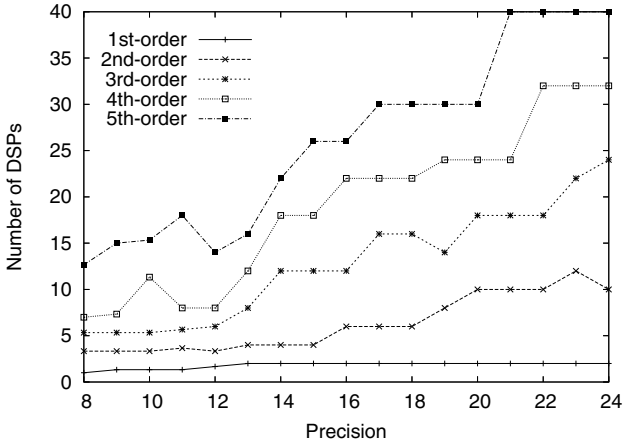


Fig. 8. Number of DSPs versus precision.

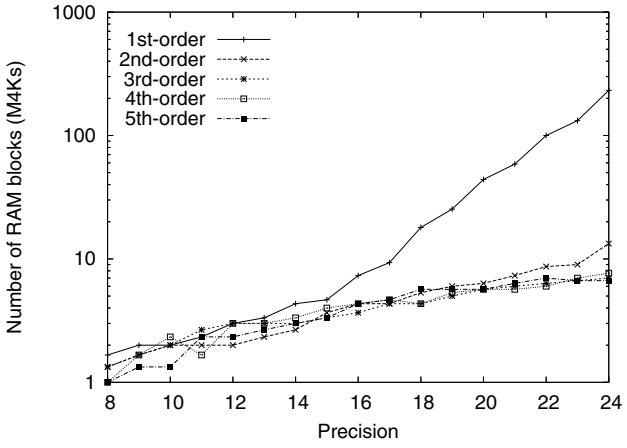


Fig. 9. Number of RAM blocks versus precision.

number of LEs. The 8 to 15-bit precision 5th-order polynomials and 8 to 11-bit precision 4th-order polynomials require fewer LEs, since they require only one segment for $\cos(\pi x)$ and therefore no memory address registers for the LUT cascade and the coefficients table. Note that when the number of segments is one, coefficients of the polynomial are implemented as constant values, not memory. Since the 22 to 24-bit precision 1st-order polynomials require large LUT cascades due to the large number of segments, the number of pipeline stages for the LUT cascade increases, and therefore the number of pipeline registers (LEs) increases excessively.

Fig. 8 shows the relation between the number of DSPs (9×9-bit multipliers) and precision. This graph shows that the number of DSPs increases as the precision increases. Further, increasing the polynomial order increases the number of DSPs. For 24-bit precision, 5th-order polynomials require 20 times more DSPs than needed for 1st-order polynomials.

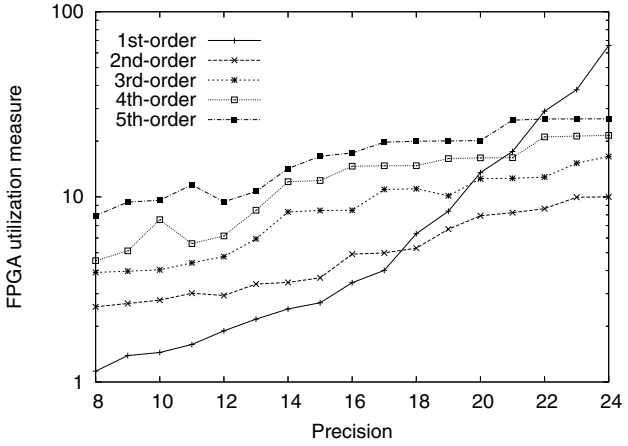


Fig. 10. FPGA utilization measure on the Stratix EP1S80F1020C5 versus precision, where all the resources are available.

Fig. 9 shows the relation between the number of RAM blocks and precision. The FPGA (EP1S80F1020C5) includes two types of RAM block, M4K and M512. But, we show only the number of M4Ks because few M512s were used for the implementations. From Fig. 9, we can see that the number of RAM blocks increases exponentially with precision. Further, increasing the polynomial order can reduce the number of RAM blocks. For 24-bit precision, a 5th-order polynomial requires only 3% of the RAM blocks required by a 1st-order polynomial.

From these results, we can see that by changing the polynomial order, we can change the amount of FPGA resources required by NFGs.

6.3. FPGA Utilization Measure

We determine the optimum polynomial order given the precision and the available hardware resources using the FPGA utilization measure.

Fig. 10 shows the relation between FPGA utilization measure and precision when all the resources on the Stratix EP1S80F1020C5 are available for a single NFG. This FPGA consists of 79,040 LEs, 176 DSPs, 364 M4Ks, and 767 M512s. From Fig. 10, we can see that for low-precision (up to 17 bits), the 1st-order polynomials yield the smallest FPGA utilization measure, and for high-precision (18 to 24 bits), the 2nd-order polynomials yield the smallest FPGA utilization measure. We view this result as very surprising, especially since the number of segments decreases significantly, when the order of the polynomial increases.

We assume a situation in which only 10% of LEs, M4Ks, and M512s, and 100% of DSPs are available. Fig. 11 shows the FPGA utilization measure for this case. From Fig. 11, it follows that, for precisions of 13 bits or less, 1st-order polynomials yield the smallest FPGA utilization measure.

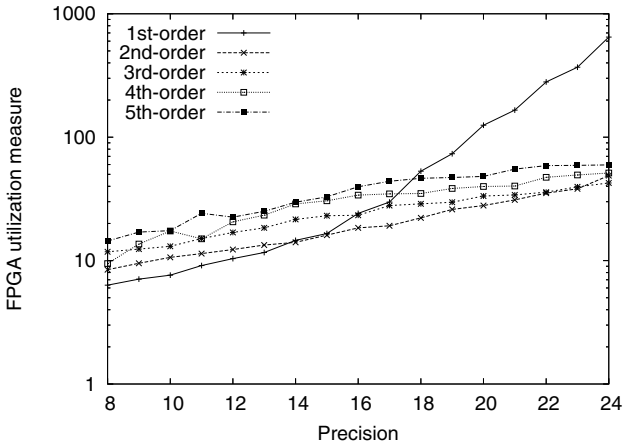


Fig. 11. FPGA utilization measure on the Stratix EP1S80F1020C5, where 10% of LEs and RAM blocks, and 100% of DSPs are available.

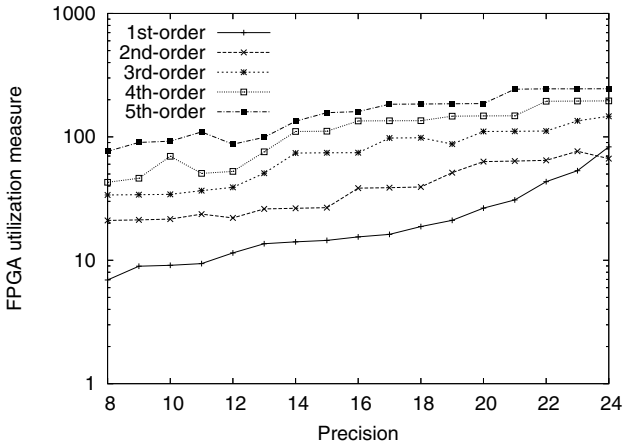


Fig. 12. FPGA utilization measure on the Stratix EP1S80F1020C5, where 10% of LEs and DSPs, and 100% of RAM blocks are available.

For 14 to 23-bit precision, 2nd-order polynomials yield the smallest FPGA utilization measure. And, for 24-bit precision, 3rd-order polynomials yield the smallest FPGA utilization measure. Note that for precisions higher than 20 bits, the 1st-order polynomials cannot be implemented in the FPGA due to insufficient RAM blocks. Fig. 12 shows the FPGA utilization measure, where only 10% of LEs and DSPs, and 100% of RAM blocks are available. From Fig. 12, we can see that for up to 23-bit precision, the 1st-order polynomials; and for 24-bit precision, the 2rd-order polynomials yield the smallest FPGA utilization measure. Note that the 3rd, 4th, and 5th-order polynomials cannot be implemented in the FPGA due to insufficient DSPs.

In order to understand how a reduction over all resources affects the realization, we implemented the NFGs on a low-cost FPGA, the Cyclone II (EP2C5F256C6, the smallest

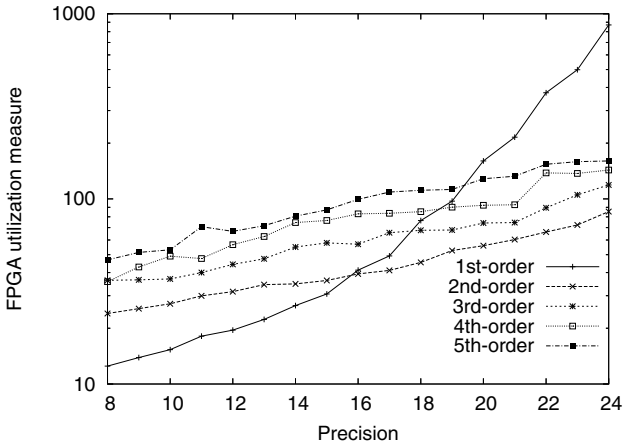


Fig. 13. FPGA utilization measure on the smallest Cyclone II EP2C5F256C6 versus precision.

device in the Cyclone II family: $4,608$ LEs, 26 DSPs, 26 M4Ks, 0 M512s). Fig. 13 shows results for the Cyclone II. For high-precision, the 1st-order polynomials deplete the RAM blocks in the Cyclone II. On the other hand, the 4th and 5th-order polynomials deplete the DSPs. Fig. 13 shows that, for up to 15-bit precision, 1st-order polynomials yield the smallest FPGA utilization measure. On the other hand, for 16 to 24-bit precision, 2nd-order polynomials yield the smallest FPGA utilization measure.

These experiments show that there is limited use for 4th and 5th-order polynomials. However, from Fig. 6, we conjecture that for higher-precision than 24-bit, 4th and 5th-order polynomials will be useful to reduce the memory size. Unfortunately, we could not verify that because of the precision of our NFG synthesis tool developed by C language.

7. Conclusion and Comments

We have presented NFGs based on k-th order polynomial approximation for $(k+1)$-times differentiable functions. To generate the most efficient NFGs, we introduced the *FPGA utilization measure*. Experimental results showed that: 1) For some kinds of numerical functions, the existing methods based on uniform segmentation cannot always reduce the memory size, even if the higher-order polynomials are used. On the other hand, our method can flexibly change the amount of hardware resources required by NFGs for a wide range of functions by changing the polynomial order k. 2) When all hardware resources in an FPGA can be used for a single NFG, for low accuracies (up to 17 bits), 1st order polynomials produce the most efficient FPGA implementation. On the other hand, for high accuracies (18 to 24 bits), 2nd order polynomials produce the most efficient FPGA implementation. Even if the amount of hardware resources is constrained, we can find the optimum polynomial order for the precision of NFG and the resource constraints.

Currently, we are developing the synthesis system that automatically generates an NFG based on the optimum polynomial order k from a given amount of hardware resources. In this system, an accurate estimate of hardware resources required by NFG is important.

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, and NSA Contract RM A-54, and the Ministry of Education, Culture, Sports, Science, and Technology (MEXT), Grant-in-Aid for Young Scientists (B), 18700048, 2007.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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)*, Leuven, Belgium, pp. 221–230, 2004.

[7] 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.

[8] 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)*, Yokohama, Japan, pp. 80–85, 2004.

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] D.-U. Lee, W. Luk, J. Villasenor, and P. Y. K. Cheung, "A Gaussian noise generator for hardware-based simulations," *IEEE Trans. on Comp.*, Vol. 53, No. 12, pp. 1523–1534, Dec. 2004.

[15] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk, "Optimizing hardware function evaluation," *IEEE Trans. on Comp.*, Vol. 54, No. 12, pp. 1520–1531, Dec. 2005.

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

[17] G. W. Morris, G. A. Constantinides, and P. Y. K Cheung, "Using DSP blocks for ROM replacement: a novel synthesis flow," *Proc. Inter. Conf. on Field Programmable Logic and Applications (FPL'05)*, Tampere, Finland, pp. 77–82, Aug. 2005.

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

[19] 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.

[20] S. Nagayama and T. Sasao, "On the optimization of heterogeneous MDDs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol. 24, No. 11, pp. 1645–1659, Nov. 2005.

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

[22] 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.

[23] 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)*, Lake Tahoe, CA, pp. 225–230, June 12–15, 2001.

[24] 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.

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

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

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

[28] 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.

[29] 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.

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