Non-uniform Segmentation for Hardware Function Evaluation

Dong-U Lee[1], Wayne Luk[1], John Villasenor[2] and Peter Y.K. Cheung[3]

[1] Department of Computing, Imperial College, London, UK
{dong.lee, wl}@ic.ac.uk
[2] Electrical Engineering Department, University of California, Los Angeles, USA
villa@icsl.ucla.edu
[3] Department of EEE, Imperial College, London, UK
p.cheung@ic.ac.uk

Abstract. This paper presents a method for evaluating functions in hardware based on polynomial approximation with non-uniform segments. The novel use of non-uniform segments enables us to approximate non-linear regions of a function particularly well. The appropriate segment address for a given function can be rapidly calculated in run time by a simple combinational circuit. Scaling factors are used to deal with large polynomial coefficients and to trade precision with range. Our function evaluator is based on first-order polynomials, and is suitable for applications requiring high performance with small area, at the expense of accuracy. The proposed method is illustrated using two functions, $\sqrt{-\ln(x)}$ and $\cos(2\pi x)$, which have been used in Gaussian noise generation.

1 Introduction

The evaluation of functions is often the performance bottleneck of many compute-bound applications. Examples of these functions include elementary functions such as $\ln(x)$ or $\sqrt{x}$, and compound functions such as $\sqrt{-\ln(x)}$ or $\tan^2(x)+1$. Computing these functions quickly and accurately is a major goal in computer arithmetic; software implementations are often too slow for numerically intensive or real-time applications. The performance of such applications depends on the design of a hardware function evaluator. Advanced FPGAs enable the development of low-cost and high-speed function evaluation units, customizable to particular applications. The principal contribution of this paper is a fast and efficient hardware function evaluator using polynomial approximations. The key novelties of our work include:

- a method for polynomial approximations with non-uniform segments;
- hardware architecture and implementation of the proposed method;
- evaluation of this method with a logarithmic function and a cosine function.

The rest of this paper is organized as follows. Section 2 covers background material and previous work. Section 3 explains our segmentation technique. Section 4 describes the hardware architecture. Section 5 presents a method for determining the placement of segment boundaries. Section 6 discusses evaluation and results, and Section 7 offers conclusion and future work.

2 Background

Polynomial approximation [13], [14] involves approximating a continuous function f with one or more polynomials p of degree n on a closed interval $[a,b]$. The aim is to minimize a distance $\|p-f\|$. There are two kinds of approximations: least squares approximations that minimize the average error, and least maximum approximations that minimize the worst-case error [15]. In both cases, the aim is to minimize a distance $\|p - f\|$. For least squares approximations, that distance is:

$$\|p - f\|_2 = \sqrt{\int_a^b w(x)(f(x) - p(x))^2 dx}, \tag{1}$$

where w is a continuous weight function for selecting parts of $[a, b]$ where we want the approximation to be more accurate. For least maximum (minimax) approximations, the distance is:

$$\|p - f\|_\infty = \max_{a \leq x \leq b} |f(x) - p(x)|. \tag{2}$$

Our work is based on minimax polynomial approximations, which involve minimizing the worst-case error. Since we are interested in fixed-point number representation in our work, we will be concerned with the worst-case absolute errors. A recent study of minimax polynomial approximation on FPGAs can be found [21].

Much of the work on function evaluation is generally concerned with producing highly accurate approximation with complex designs. Instead, we will focus on applications that require very high speed and small area but not high accuracy. Examples of such applications include Gaussian noise generation [3] and belief propagation in LDPC decoding [20]. We will focus in this paper on first-order polynomials of the form $p(x) = c_1 \times x + c_0$, where c_1 is the gradient and c_0 is the y-intercept, which can be computed by two table lookups, a multiplication and an addition.

Previous work on polynomial approximations involves equally sized segments [4], [5], [6], [7], [8], [9], [10], [11], [12]. Approximations using such uniform segments are suitable for functions with linear regions, but they can be inefficient for non-linear functions. It is desirable to choose the boundaries of the segments to cater for the non-linearities of the function. Highly non-linear regions may need smaller segments than linear regions. This approach minimizes the amount of storage required to approximate the function, leading to more compact and efficient designs.

3 Function Evaluation based on Non-uniform Segmentation

The interval of approximation $[a, b]$ is divided into a set of sub-intervals, called segments. The best-fit straight line, in a minimax sense, to each segment is found. A lookup table is used to store the coefficients for each line segment, and the functions can then be evaluated using a multiplier and an adder to calculate the linear approximation [1].

Using well-known methods that compute elementary functions such as CORDIC [2], the evaluation of compound functions is a multi-stage process. Consider the evaluation of the function $\sqrt{-\ln(x)}$ over the interval $(0, 1]$. Using CORDIC, the computation of this function is a two-stage process: the logarithm of x followed by the square root. With our approach, we look at the entire function over the given domain, and therefore we do not need to have two stages.

As shown in Figure 1, the greatest non-linearities of the function $\sqrt{-\ln(x)}$ occur in the regions close to zero and one. If uniform segments are used, a large number of small segments would be required to get accurate approximations in the non-linear regions. However, in the middle part of the curve where it is relatively linear, accurate approximation can be obtained using relatively few segments. It would be efficient to use small segments for the non-linear regions, and large segments for linear regions. Arbitrary-sized segments would enable us to have the least error for a given number of segments; however, the hardware to calculate the segment address for a given input can be complex. Our objective is to provide near arbitrary-sized segments with a simple circuit to find the segment address for a given input.

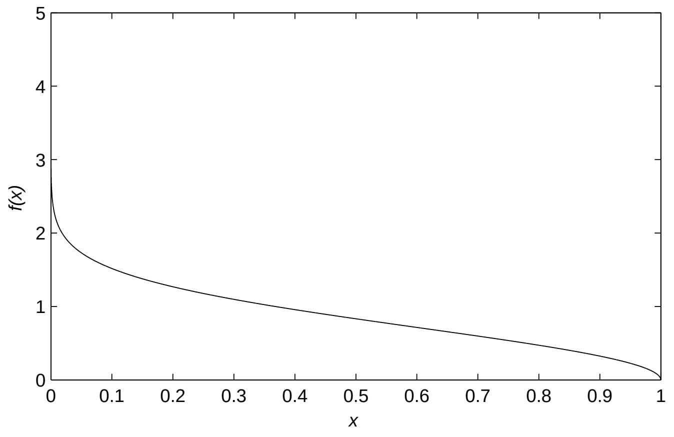

Fig. 1. $\sqrt{-\ln(x)}$ over $(0, 1]$.

We have developed a novel method which can construct piecewise linear approximation such that: (a) the segment lengths used in a given region depends on the local linearity, with more segments deployed for regions of higher non-linearity; and (b) the boundaries between segments are chosen such that the task of identifying which segment to use for a given input can be rapidly performed. The proposed method consists of five steps.

1. Determine optimal placement of segment boundaries (see Section 5) – this would include dividing into regions such that in each region the function either monotonically increases or decreases.
2. For a non-linear region, if the non-linearity is monotonically increasing, then increase segment size by a factor of two or more at each step; if the non-linearity is monotonically decreasing, then reduce segment size by a factor of two or more at each step.
3. The segment addresses can be obtained by computing the prefixes [16] with a simple combinational or pipelined circuit.
4. If necessary, divide the function into several intervals, then apply step 1–3 (see the function $\cos(2\pi, x)$ in Section 6).
5. If necessary, repeat the above steps with higher-order terms.

As an example to illustrate our approach, consider approximating $\sqrt{-\ln(x)}$ with an 8-bit input (Figure 1). Using the traditional approach, the most-significant bits of x are used to index the uniform segments. For instance if the most-significant four bits are used, 16 uniform segments are used to approximate the function. Using our approach, it is possible to use small segments for non-linear regions (regions near 0 and 1), and large segments for linear regions (regions around 0.5). The idea is to use segments that grow by a factor of two from 0 to 0.5, and segments that shrink by a factor of two from 0.5 to 1 in the x-axis of Figure 1. We use segment boundaries at locations 2^{n-8} and $1 - 2^{-n}$ where $0 \leq n < 8$. Up to 14 segments can be formed this way. A circuit based on prefix computation can be used for calculating segment addresses (Figure 2) for a given input x. It checks the number of leading zeros and ones to work out the segment address. A cascade of OR gates is used for segments that grow by factors of two, and a cascade of AND gates is used for segments that shrink by factors of two; these circuits can be pipelined and a circuit with shorter critical path but requiring more area can be used [16]. Note that the choice of segments does not have to be factors of two, it could be more. The appropriate taps are taken from the cascades depending on the choice of the segments and are added to work out the segment address. In Figure 2, the maximum available taps are taken, giving 14 segment addresses. Some taps would not be taken if the segments grow or shrink by more than a factor of two. It can be seen that the critical path of this circuit is the path from x_6 or x_7 to the output of the adder. By introducing pipeline registers between the gates, higher throughput can be easily achieved.

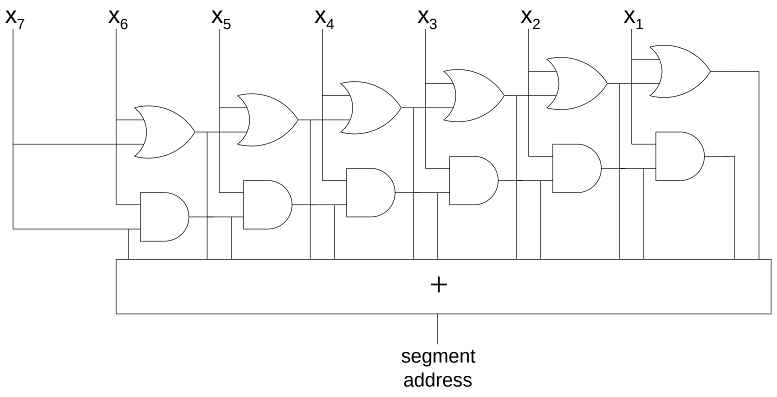

Fig. 2. Circuit to calculate the segment address for a given input x. The adder counts the number of ones in the output of the two prefix circuits.

When approximating $\sqrt{-\ln(x)}$ with 32-bit inputs based on polynomials of the form $p(x) = c_1 \times x + c_0$, the gradient of the steepest part of the curve is in the order of 10^8, thus large multipliers would be required. To overcome this problem, we use scaling factors of multiples of two to reduce the magnitude of the gradient, essentially trading precision for range. This is appropriate since the larger the gradient, the less important precision becomes. The use of scaling factors provides the user the ability to control the precision for both c_1 and c_0, resulting in variation of the size of the multiplier and adder. Hence for each segment four coefficients are stored: c_1 and its scaling factor, c_0 and its scaling factor.

It is also possible to divide the input interval into uniform or non-uniform intervals, and have uniform or non-uniform segments inside each interval. In this case, the most-significant bits are used to address the intervals, and the least-significant bits are used to address the segments inside each interval. It can be seen that one can have any number of nested combinations of uniform and non-uniform segments. This hybrid combination of nested uniform and non-uniform segments provides a flexible way to choose the segment boundaries. Currently, this segmentation step is done by hand, which is slow and far from optimal. A possible approach to automate this step is discussed in Section 5.

4 Hardware Architecture

The architecture of our function evaluator shown in Figure 3 is based on polynomials of the form $p(x) = c_1 \times x + c_0$. The most-significant bits are used to select the interval, and the least-significant bits are passed through the segment address calculator which calculates the segment address within the interval. The design shown is developed for the common cases, and has been used in the examples of this paper. For other cases, one could divide the input bits into more than two parts and apply the segment address calculation depending on whether the parts use uniform or non-uniform segments.

The ROM outputs the four coefficients for the chosen interval and segment. c_1 is multiplied by the input x and c_s_1 is used to scale the output. The scaling circuit involves shifters, which increase or decrease the value by powers of two. This scaled multiplication value is added to the scaled c_0 coefficient to produce the final result.

For high throughput applications, the segment address calculator, the multiplier and the adder can be pipelined. For typical applications targeting FPGAs, the ROM would be small and could be implemented on-chip using distributed RAM or block RAM. Often

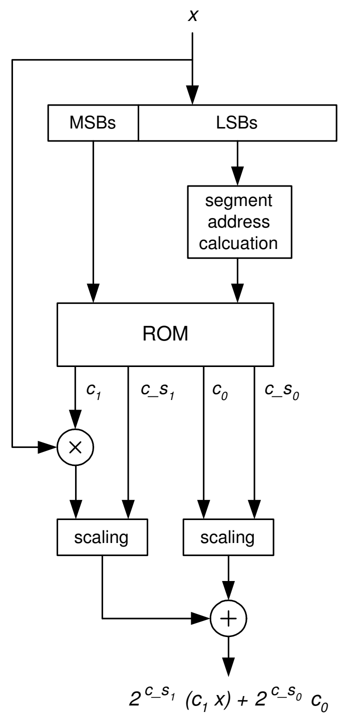

Fig. 3. Our function evaluator architecture.

the multiplier would be the part taking up a significant portion of the area. Therefore it is important to minimize the multiplier size by finding out the minimum bit width for the coefficient c_1. Also recent FPGAs, such as Xilinx Virtex-II devices, provide dedicated hardware resources for multiplication which can benefit the proposed architecture.

5 Placement of Segment Boundaries

Let f be a continuous function on $[a, b]$, and let an integer $m \geq 2$ specify the number of contiguous intervals into which $[a, b]$ has been partitioned: $a = u_0 \leq u_1 \leq ... \leq u_m = b$. Let n_i and $d_i (i = 1, ..., m)$ be non-negative integers and let P_i denote the set of rational functions p_i whose numerators and denominators are polynomials of degrees less or equal to n_i and d_i, respectively. For $i = 1, ..., m$, define

$$h_i(u_{i-1}, u_i) = \min_{p_i \in P_i} \max_{u_{i-1} \leq x \leq u_i} |f(x) - p_i(x)|. \tag{3}$$

Let $\mu = \mu(u) = \max_{1 \leq i \leq m} h_i(u_{i-1}, u_i)$. Lawson states in his paper [18] that the segmented rational minimax approximation problem is that of minimizing μ over all partitions u of $[a, b]$. It can be shown that if the error norm is a non-decreasing function of the length of the interval of approximation, that the function to be approximated is continuous and that the goal is to minimize the maximum error norm on each interval, then a balanced error solution is optimal; the term “balanced error” means that the error norms on each interval are equal.

Pavlidis and Maika present an iterative scheme for segmentation in their paper [19] which results in a suboptimal balanced error solution. The scheme is based on an iteration of the form

$$u_m^{k+1} = u_m^k + c(e_{m+1}^k - e_m^k), \quad m = 1, ..., n-1. \tag{4}$$

Here u_m^k is the value of the m-th point and the k-th iteration, e_m^k is the error on $(u_{m-1}^k, u_m^k]$ and c is an appropriate small positive number. It can be shown that for sufficiently small c the scheme converges to a solution [19]. In this algorithm, the number of segments is fixed and this determines the maximum error. However in many cases, it may be more useful to fix the accuracy desired and let the number of segments vary. Starting from a or b one could apply polynomial approximation in small increments, until the desired accuracy is reached. Then start a new segment from that point.

Once the segment boundaries have been found by using one of the two approaches above, the next step is to match the boundaries based on our addressing scheme as close to the suboptimum ones as possible. As discussed in Section 3, our addressing scheme is based on nested uniform and non-uniform segments. By carefully using these combinations of segments, it is possible to get a close approximation to the suboptimum segment boundaries. Our aim is to enable the user to input constraints such as maximum error norm and to apply the segmentation automatically to produce lookup tables and the corresponding circuits such as the one shown in Figure 3. A possible approach of such an automated method is shown in Figure 4.

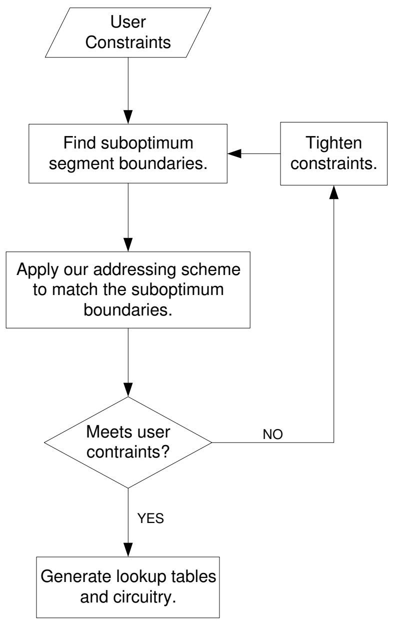
User Constraints
Find suboptimum segment boundaries.
Tighten constraints.
Apply our addressing scheme to match the suboptimum boundaries.
Meets user contraints?
NO
YES
Generate lookup tables and circuitry.

Fig. 4. Steps for automating segmentation.

6 Evaluation and Results

Our function evaluator has been successfully implemented for the Gaussian noise generator presented in [3]. Three functions are approximated: $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and $\sin(2\pi x)$ over $[0, 1]$. 32-bit inputs are used for $\sqrt{-\ln(x)}$ and 16-bit inputs are used for $\cos(2\pi\, x)$ and $\sin(2\pi x)$.

We first consider the function $\sqrt{-\ln(x)}$. As stated earlier, the greatest non-linearities of this function occur in the regions close to zero and one. To be consistent with the change in linearity, we use line segment locations to boundaries at locations 2^{n-32} for $0 < x \leq 0.5$, and $1 - 2^{-n}$ for $0.5 < x \leq 1$, where $0 \leq n < 32$. A total of 59 segments are used to approximate this function as shown in Figure 5. Since $\sqrt{-\ln(x)}$ approaches infinity for x

values close to zero, the smallest x value is $1/2^{32}$, resulting in a maximum output value of around 4.7.

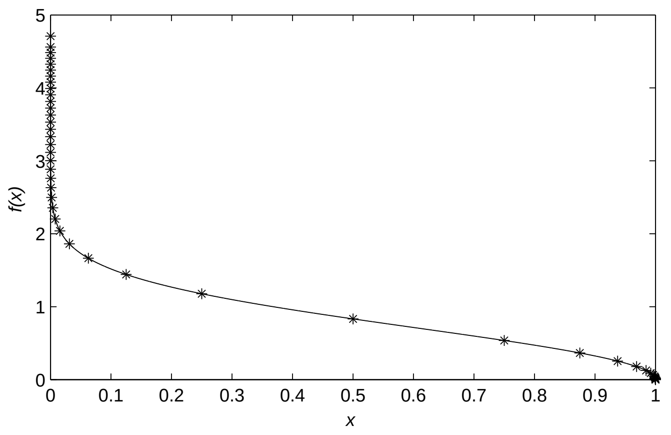

Fig. 5. The segments used to approximate $\sqrt{-\ln(x)}$ with 32-bit inputs. The asterisks indicate the segment boundaries of the linear approximations.

The maximum absolute error of this approximation is 0.020. However this is the case only if we have infinite precision for the coefficients, which is not realistic. Multipliers take significant amount of resources on FPGAs, therefore the coefficients for the gradient should be as small as possible. Tests are carried out to find the optimum number of bits for the gradient coefficients that provides the least absolute error. Figure 6 shows how the maximum absolute error varies with the number of bits used for the gradient of $\sqrt{-\ln(x)}$. The figure indicates that six bits are sufficient to give a maximum absolute error of 0.031. The approximation should differ from the true value by less than one unit in the last place (ulp) [17]; the least significant bit of the fraction of a number in its standard representation is defined to be the last place. With this error, it is sufficient to give an output accuracy of eight bits (three bits for integer and five for fraction). If uniform segments are used, small segment size would be needed in order to cope with the highly non-linear parts of the curve. In fact, one would require around 617 million segments to get the same maximum absolute error with uniform segments. This is a good example to demonstrate the effectiveness of our non-uniform approach. It is clear that our approach works well especially for functions with exponential behavior.

To evaluate the functions $\cos(2\pi x)$ and $\sin(2\pi x)$, due to the symmetry of the sine and cosine functions, only the input range $[0, 1/4]$ for $\cos(2\pi x)$ needs to be approximated [15]. The specific axis-partitioning technique for $\sqrt{-\ln(x)}$ is unsuitable for $\cos(2\pi x)$, since the non-linearities of the two functions are different. If the same technique is used, there would be many unnecessary segments near the beginning and end of the curve, and not enough segments in the middle regions. As before we consider both the local linearity of the curve, and the computational concerns with respect to choosing specific segment boundary locations, leading to the approximations shown in Figure 7. The curve is divided into four uniform intervals and within each interval, non-uniform segmentation is applied. Note that for each interval, not all taps are taken from the segment address calculator. We use a total of 21 segments to approximate this function.

With finite precision on the coefficients, the maximum absolute error of this approximation is 0.0035, which is sufficient to give an output accuracy of eight bits (all eight bits for fraction). Using uniform segments, the same error can be obtained with a slightly larger number of segments; this is because the curve does not have high non-linearities.

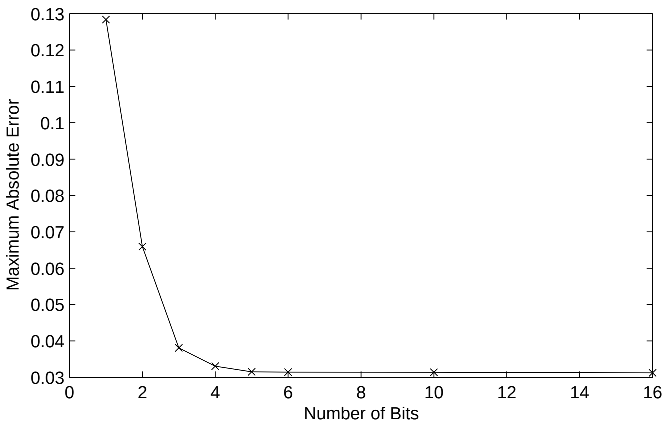

Fig. 6. Variation of function approximation error with number of bits for the gradient of $\sqrt{-\ln(x)}$.

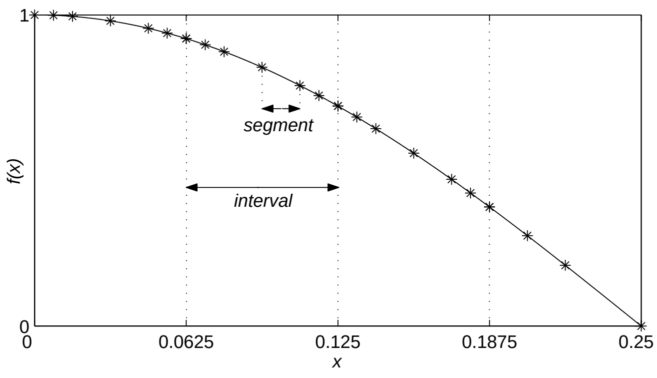

Fig. 7. Approximation for $\cos(2\pi x)$ over $[0, 1/4]$. The asterisks indicate the segment boundaries of the linear approximations.

Table 1 shows a comparison of the number of segments for the two functions for non-uniform and uniform segmentation in order to achieve the same worst-case error. Note that for uniform segmentation, the number of segments needs to be a power of two. This is because the most-significant n bits are used for addressing. For instance, the actual number of uniform segments needed for the $\sqrt{-\ln(x)}$ function is 617 million, but 1 billion segments are used which is the next power of two (2^{30}). We do not have this kind of a restriction with our non-uniform addressing scheme. The table also shows the number of bits used for each coefficient in the look-up tables. The lookup tables for the three functions $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and $\sin(2\pi x)$ have a total size of just 3504 bits. With such small lookup table size, all the coefficients can be stored on-chip for fast access.

Table 1. Second column shows the comparison of the number of segments for non-uniform and uniform segmentation. Third column shows number of bits used for the coefficients to approximate the $\sqrt{-\ln(x)}$ and $\cos(2\pi\, x)$ functions.

function	non-uniform	uniform	c_1	c_s_1	c_0	c_s_0
$\sqrt{-\ln(x)}$	59	1 billion	6	5	32	5
$\cos(2\pi\, x)$	21	32	8	4	16	4

The function evaluators for the three functions are written using the Handel-C hardware compiler from Celoxica [25], and are mapped and tested on a Xilinx Virtex-II XC2V4000-6 device [24]. The design occupies 1864 slices, four block multipliers and two block RAMs, and takes up around 7% of the device. A fully pipelined version of our design operates at 133 MHz with a latency of 14 clock cycles, and the function evaluators are capable of 133 million operations per second; the completion time for each input is given by 14 / 133 million = 105 ns. The design has also been implemented on a low cost Xilinx Spartan-IIE XC2S300E-7, which occupies 70% of the chip and is capable of 62 million operations per second. Our hardware implementations have been compared with software implementations (Table 2). The Virtex-based FPGA implementation is 158 times faster than the Athlon-based PC in terms of throughput, and 11 times faster in terms of completion time.

Table 2. Performance comparison: computation of $\sqrt{-\ln(x)}$, $\cos(2\pi x)$ and $\sin(2\pi x)$. All PCs are equipped with 512MB DDR RAM. The XC2V4000-6 FPGA belongs to the Xilinx Virtex-II family, while the XC2S300E-7 belongs to the Xilinx Spartan-IIE family. The software implementations are written in C generating single precision floating point numbers, and are compiled with the GCC 3.3 compiler [26].

platform	clock speed	latency	area	throughput	completion time
	(MHz)	(clock cycles)	(slices)	(operations / second)	(ns)
XC2V4000-6 FPGA	133	14	1864	133 million	105
XC2S300E-7 FPGA	62	14	2129	62 million	226
AMD Athlon PC	1400	-	-	0.84 million	1187
Intel Pentium 4 PC	2400	-	-	0.79 million	1261

Well-known function evaluation methods, such as SBTM [5], [6], deal with the approximation of elementary functions over a fixed input range where the function is linear. Range reduction techniques such as those presented in [22] and [23] are used to bring the input within the linear range. However, range reduction is not possible for most compound functions. Our approach caters for both non-linear and linear regions, which makes it suitable for both elementary and compound functions. Currently, our approach tends to produce small lookup table sizes with low accuracy; we hope to improve accuracy by further work on automatic segmentation.

7 Conclusion

This paper presents a novel method for evaluating functions using polynomial approximations by employing non-uniform segments. The non-uniform segments deal with the non-linearities of functions which occur frequently. A simple cascade of AND and OR gates can be used to rapidly calculate the segment address for a given input. Scaling factors are used to deal with large polynomial coefficients, trading precision with range. Two functions developed for the generation of Gaussian noise are used as examples to illustrate and to evaluate our approach. Results show the advantages of using non-uniform segments over uniform ones. Current and future work includes automating the selection of boundaries, and exploring the use of higher order polynomials for more accurate approximations. This would enable us to apply our approach to a wide range of functions and to obtain detailed comparison with other methods. We will also look at how our function evaluator can be used to speed up addition and subtraction functions in logarithmic number systems [12], which are highly non-linear functions.

Acknowledgment

The authors thank Jun Jiang and Shay Ping Seng for their assistance. The support of Celoxica Limited, Xilinx Inc., the U.K. Engineering and Physical Sciences Research Council (Grant number GR/N 66599, GR/R 55931 and GR/R 31409), and the U.S. Office of Naval Research is gratefully acknowledged.

References

1. O. Mencer, N. Boullis, W. Luk and H. Styles, "Parameterized function evaluation for FPGAs", *Field-Programmable Logic and Applications*, LNCS 2147, pp. 544–554, 2001.
2. J.E. Volder, "The CORDIC trigonometric computing technique", *IEEE Trans. on Elec. Comput.*, vol. EC-8, no. 3, pp. 330–334, 1959.
3. D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "A hardware Gaussian noise generator for channel code evaluation", *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach.*, 2003.
4. D. Das Sarma and D.W. Matula, "Faithful bipartite rom reciprocal tables", *Proc. 12th IEEE Symp. on Comput. Arith.*, pp. 17–28, 1995.
5. M.J. Schulte and J.E. Stine, "Symmetric bipartite tables for accurate function approximation", *Proc. 13th IEEE Symp. on Comput. Arith.*, vol. 48, no. 9, pp. 175–183, 1997.
6. M.J. Schulte and J.E. Stine, "Approximating elementary functions with symmetric bipartite tables", *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 842-847, 1999.
7. J.A Pineiro, J.D. Bruguera and J.M. Muller, "Faithful powering computation using table look-up and a fused accumulation tree", *Proc. 15th IEEE Symp. on Comput. Arith.*, 2001.
8. J. Cao, B.W.Y. We and J. Cheng, " High-performance architectures for elementary function generation", *Proc. 15th IEEE Symp. on Comput. Arith.*, 2001.
9. V.K. Jain, S.A. Wadecar and L. Lin, "A universal nonlinear component and its application to WSI", *IEEE Trans. Components, Hybrids and Manufacturing Tech.*, vol. 16, no. 7, pp. 656–664, 1993.
10. H. Hassler and N. Takagi, "Function evaluation by table look-up and addition", *Proc. of the IEEE 12th Symp. on Comp. Arith.*, pp. 10–16, 1995.
11. J. Detrey and F. de Dinechin, "Multipartite tables in JBits for the evaluation of functions on FPGAs", *Proc. IEEE Int. Parallel and Distributed Processing Symp.*, 2002.
12. D.M. Lewis, "Interleaved memory function interpolators with application to an accurate LNS arithmetic unit", *IEEE Trans. Comput.*, vol. 43, no. 8, pp. 974–982, 1994.
13. J.F. Hart et al., *Computer Approximations*, Wiley, 1968.
14. J.R. Rice, *The Approximation of Functions*, vol. 1,2, Addison-Wesley, 1964, 1969.
15. J.M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser Verlag AG, 1997.
16. R.E. Ladner and M.J. Fischer, "Parallel prefix computation", *JACM*, vol. 27, no. 4, pp. 831–838, 1980.
17. I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall, 1993.
18. C.L. Lawson, "Characteristic properties of the segmented rational minimax approximation problem", *Numer. Math.*, vol. 6, pp. 293–301, 1964.
19. T. Pavlidis and A.P Maika, "Uniform piecewise polynomial approximation with variable joints", *Journal of Approximation Theory*, vol. 12, pp. 61–69, 1974.
20. C. Jones, E. Vallés, C. Wang, M. Smith, R. Wesel and J. Villasenor, "High throughput Monte Carlo simulation for error floor testing in capacity achieving channel codes", *Proc. IEEE Symp. on Field-Prog. Cust. Comput. Mach.*, 2003.
21. N. Sidahao, G.A. Constantinides and P.Y.K. Cheung, "Architectures for function evaluation on FPGAs", *Proc. IEEE Int. Symp. on Circ. and Syst.*, 2003.
22. J.S. Walther, "A unified algorithm for elementary functions", *Proc. Spring Joint Comput. Conf.*, 1971.
23. N.W. Cody and W. Waite, *Software Manual for the Elementary Functions*, Prentice-Hall, 1980.
24. Xilinx Inc., *Virtex-II User Guide v1.5*, 2002.
25. Celoxica Limited, *Handel-C Language Reference Manual*, ver. 3.1, document no. RM-1003-3.0, 2002.
26. GNU Project, *GCC 3.3 Manual*, http://gcc.gnu.org, 2003.