Hierarchical Segmentation Schemes for Function Evaluation

Dong-U Lee and Wayne Luk
Department of Computing
Imperial College
London
United Kingdom
{dong.lee, w.luk}@ic.ac.uk

John Villasenor
Electrical Engineering Department
University of California
Los Angeles
USA
villa@icsl.ucla.edu

Peter Y.K. Cheung
Department of EEE
Imperial College
London
United Kingdom
p.cheung@ic.ac.uk

Abstract

This paper presents a method for evaluating functions based on piecewise polynomial approximation with a novel hierarchical segmentation scheme. The use of a novel hierarchy scheme of uniform segments and segments with size varying by powers of two enables us to approximate non-linear regions of a function particularly well. This partitioning is automated: efficient look-up tables and their coefficients are generated for a given function, input range, order of the polynomials, desired accuracy and finite precision constraints. We describe an algorithm to find the optimum number of segments and the placement of their boundaries, which is used to analyze the properties of a function and to benchmark our approach. Our method is illustrated using three non-linear compound functions, $\sqrt{-\log(x)}$, $x\log(x)$ *and a high order rational function. We present results for various operand sizes between 8 and 24 bits for first and second order polynomial approximations.*

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 $\log(x)$ and $\sqrt{x}$, and compound functions such as $\sqrt{-\log(x)}$ and $\tan^2(x)+1$. Computing these functions quickly and accurately is a major goal in computer arithmetic.

The evaluation of elementary functions has received significant interest [8]. In contrast, there has been little attention on the efficient approximation of compound functions for special purpose applications. Examples of such applications include Gaussian noise generation [6], channel coding [3] and image registration [11]. In principle, these compound functions can be approximated by splitting them into several elementary functions, but this approach would result in long delay, propagation of rounding errors and possibilities of catastrophic cancellations. Range reduction [8] is not feasible for compound functions, so highly non-linear regions of a function need to be approximated. Since we are looking at the entire function over a given input range, the advantages of our method increase significantly as compound functions become more complex. We illustrate our method with the following three functions:

$$f_1 = \sqrt{-\log(x)} \quad (1)$$

$$f_2 = x\log(x) \quad (2)$$

$$f_3 = \frac{0.0004x + 0.0002}{x^4 - 1.96x^3 + 1.348x^2 - 0.378x + 0.0373} \quad (3)$$

where x is an n-bit number over $[0, 1)$ of the form $0.x_{n-1}..x_0$. The function f_1 is used in the Box-Muller algorithm for the generation of Gaussian noise [6], and f_2 is commonly used for entropy calculation such as mutual information computation in image registration [11]. Note that the functions f_1 and f_2 cannot be computed for $x = 0$, therefore we approximate these functions over $(0, 1)$ and generate an exception when $x = 0$. In this paper, we implement an n-bit in n-bit out system. However, the position of the decimal (or binary) point in the input and output formats can be different in order to maximize the precision that can be described.

The principal contribution of this paper is a systematic method for producing fast and efficient hardware function evaluators for both compound and elementary functions using piecewise polynomial approximations with a hierarchical segmentation scheme. The novelties of our work include:

- an algorithm for locating optimum segment boundaries given a function, input interval and maximum error;
- a scheme for piecewise polynomial approximations with a hierarchy of segments;
- evaluation of this method with three compound functions;

- hardware architecture and implementation of the proposed method.

The rest of this paper is organized as follows: Section 2 covers background material and previous work. Section 3 explains how our algorithm finds the optimum placement of the segments. Section 4 presents our hierarchical segmentation scheme. Section 5 describes our hardware architecture. Section 6 discusses evaluation and results, and Section 7 offers conclusions and future work.

2 Background

Many applications including digital signal processing, computer graphics and scientific computing require the evaluation of mathematical functions. Applications that do not require high precision, often employ direct table look-ups. However, this becomes impractical for precisions higher than a few bits, because the size of the table is exponential in the input size.

CORDIC has been a popular method for evaluating functions, involving only shift and add operations. However, it has an execution time which is linearly proportional to the number of operands, and is not suitable for applications requiring high accuracy and speed .

Recently, table look-up and addition methods have attracted significant attention. Table look-up and addition methods use two or more parallel table look-ups followed by multi-operand addition. Such methods include the symmetric bipartite table method (SBTM) [12] and the symmetric table addition method (STAM) [13]. These methods exploit the symmetry of the Taylor approximations and leading zeros in the table coefficients to reduce the look-up table size. Although these methods yield in significant improvements in table size over direct look-ups, they can be inefficient for functions that are highly non-linear.

Piecewise polynomial approximation [10] which is the method we use in this paper, involves approximating a continuous function f with one or more polynomials p of degree d on a closed interval $[a, b]$. The polynomials are of the form

$$p(x) = c_d x^d + c_{d-1} x^{d-1} + ... + c_1 x + c_0 \quad (4)$$

and with Horner's rule, this becomes

$$p(x) = ((c_d x + c_{d-1})x + ...)x + c_0 \quad (5)$$

where x is the input. The aim is to minimize a distance $||p-f||$. Our work is based on minimax polynomial approximations, which involve minimizing the maximum absolute error. The distance for minimax approximations is:

$$||p - f||_\infty = \max_{a \leq x \leq b} |f(x) - p(x)|. \quad (6)$$

The minimax polynomial is found in an iterative manner using the Remez exchange algorithm, which is often used for determining optimal coefficients for digital filters.

Approximations using uniform segments are suitable for functions with linear regions, but are inefficient for non-linear functions, especially when the function varies exponentially. 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. We use a hierarchy of uniform segments (US) and powers of two segments (P2S): segments with the size varying by increasing or decreasing powers of two.

3 Optimum Placement of Segments

This section introduces a method for computing the optimum placement of segments for function approximation. We shall use it as a reference in comparing the uniform segment method and our proposed method, as shown in Table 2 (Section 4). Let f be a continuous function on $[a, b]$, and let an integer $m \geq 2$ specify the number of contiguous segments into which $[a, b]$ has been partitioned: $a = u_0 \leq u_1 \leq ... \leq u_k = b$. Let d be a non-negative integer and let P_i denote the set of functions p_i whose polynomials are of degrees less or equal to d. For $i = 1, ..., m$, define

$$h_i(u_{i-1}, u_i) = \min_{p_i \in P_i} \max_{u_{i-1} \leq x \leq u_i} |f(x) - p_i(x)|. \quad (7)$$

Let $e_{max} = e_{max}(u) = \max_{1 \leq i \leq m} h_i(u_{i-1}, u_i)$. The segmented minimax approximation problem is that of minimizing e_{max} over all partitions u of $[a, b]$. If the error norm is a non-decreasing function of the length of the interval of approximation, 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 [5]. One class of algorithms to tackle this problem is based on the remainder formula [2] and assumes that the $(d + 1)$th derivative of f is either of fixed sign or bounded away from zero. However, in many practical cases this assumption does not hold [9]. Often, the $(d + 1)$th derivative may be zero or very small over most of $[a, b]$ except a few points where it has very large values. This precisely is the case with the non-linear functions we are approximating.

We have developed a novel algorithm to find the optimum boundaries for a given f, $[a, b]$, d, e_{max} and unit in the last place (ulp). The ulp is the least significant bit of the fraction of a number in its standard representation. For

instance, if a number has F fractional bits, the ulp of that number would be 2^{-F}. The ulp is required as an input, since the input is quantized to n bits. The MATLAB code for the algorithm is shown in Figure 1. The algorithm is based on binary search and finds the optimum boundaries over $[a, b]$. We first set $x1 = a$ and $x2 = b$ and find the minimax approximation over the interval $[x1, x2]$. If the error e of this approximation is larger than e_{max}, we set $x2 = x2/2$ and obtain the error for $[a, x2]$. We keep halving the interval of approximation until $e \leq e_{max}$. At this point we increment $x2$ by a small amount and compute the error again. This small amount is either $(abs(x2 - prev_x2))/2$ or the ulp, whichever is smaller ($prev_x2$ is the value of $x2$ in the previous iteration). When this small amount is the ulp, in the next iterations $x2$ will keep oscillating between the ideal (un-quantized) boundary. We take the $x2$ whose error e is just below e_{max} as our boundary, set $x1 = x2$ and $x2 = b$, and move on to approximating over $[x1, x2]$. This is performed until the error over $[x1, x2]$ is less than or equal to e_{max} and $x2$ has the same value as b. We can see that the boundaries up to the last one are optimum for the given ulp (the last segment is always smaller than its optimum size, as it can be seen in Figure 2 for f_2). Although our segments are not optimum in the sense that the errors of the segments are not fully balanced, we can conclude that given the error constraint e_{max} and the ulp, the placement of our segment boundaries is optimum. This is because the maximum error we obtain is less than or equal to e_{max} and this is not achievable with fewer segments.

The results of our segmentation can be used for various other applications [9] including pattern recognition, data compression, non-linear filtering and picture processing. In the ideal case, one would use these optimum boundaries to approximate the functions. However, from a hardware implementation point of view, this can be impractical. The circuit to find the right segment for a given input could be complex, hence large and slow. Nevertheless, the optimum segments give us an indication of how well a given segmentation scheme matches the optimum segmentation. Moreover, they provide information on the non-linearities of a function. Figure 2 shows the optimum boundaries for the three functions in Section 1 for 16-bit operands and second order approximations. We observe that f_1 needs more segments in the regions near 0 and 1, f_2 requires more segments near 0 and f_3 requires more segments in the two regions in the lower and upper half of the interval.

Figure 3 compares the ratio of the number of optimum segments required by first and second order approximations for 8, 12, 16, 20 and 24-bit approximations to the three functions. We can see that savings of second order approximations get larger as the bit width increases. However one should note that, whereas first order approximations involve one multiply and one add, second order approximations involve two multiplies and two adds. Therefore, there is a tradeoff between the look-up table size and the circuit complexity.

```
% Inputs: a, b, d, f, e_max, ulp
% Output: u()

x1 = a; x2 = b; m = 1; done = 0; check_x2 = 0; prev_x2 = a;
oscillating = 0; done = 0;

while (~done)
    error = minimax(f,d,x1,x2,ulp);
    if (error <= req_error)
        if (x2 == b)
            u(i) = x2;
            done = 1;
        else
            if (oscillating)
                u(m) = x2;
                prev_x2 = x2;
                x1 = x2;
                x2 = b;
                m = m+1;
                oscillating = 0;
            else
                change_x2 = abs(x2-prev_x2)/2;
                prev_x2 = x2;
                if (change_x2 > ulp)
                    x2 = x2 + change_x2;
                else
                    x2 = x2 + ulp;
                end
            end
        end
    else
        change_x2 = abs(x2-prev_x2)/2;
        prev_x2 = x2;
        if (change_x2 > ulp)
            x2 = x2 - change_x2;
        else
            x2 = x2 - ulp;
            if (check_x2 == x2)
                oscillating = 1;
            else
                check_x2 = x2;
            end
        end
    end
end
```

Figure 1. MATLAB code for finding the optimum boundaries.

4 Hierarchical Segmentation

Let a segmentation scheme $\Lambda \in \{\text{US}, \text{P2S}\}$ where US = uniform segments and P2S = powers of two segments. The proposed segment hierarchy H is of the form $\Lambda_0(\Lambda_1(...(\Lambda_{\lambda-1})))$ where λ is the number of levels in the hierarchy. This structure can be implemented in a cascaded look-up table structure, where the output of one table is used as the address of the next. Let $i = 0..\lambda$. The input x is split into $\lambda + 1$ partitions called δ_i. Let v_i denote the bit width and s_i denote the number of segments of the ith partition δ_i. Therefore, $n = \sum_{i=0}^{\lambda} v_i$, where n is the number of bits of the input x. Then s_i can be defined with the following set

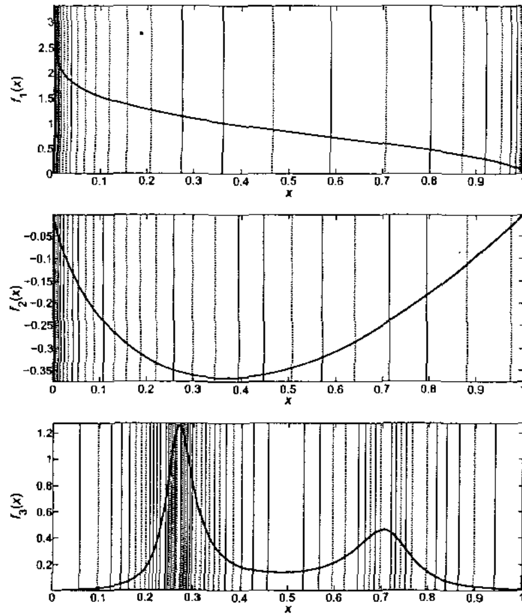

Figure 2. Optimum locations of the segments for the three functions in Section 1 for 16-bit operands and second order approximation.

of equations:

$$s_i = 2^{v_i}, \quad \text{if } \Lambda_i = \text{US} \tag{8}$$

$$s_i \leq 2v_i, \quad \text{if } \Lambda_i = \text{P2S} \tag{9}$$

For US, it is clear that 2^{v_i} segments can be formed, since uniform segments are addressed with v_i bits. However for P2S, it is not so clear why up to $2v_i$ segments can be formed.

Consider the case when Λ_0 = P2S, $n = 8$, $v_0 = 5$ and $v_1 = 3$. When $v_i = 5$ it is possible to construct 10 P2S as illustrated in Table 1. Notice that the segment sizes increase by powers of two till "**01111111**" (end of location 4) and start decreasing by powers of two from "**10000000**" (beginning of location 5) until the end. It can be seen that the maximum number of P2S that can be constructed with δ_i is $2v_i$. *Fewer segments can be obtained by omitting parts* of the table. For example with locations 0-4, one can have segments that only increase by powers of two. To compute the segment address for a given input δ_0, we need to detect the leading zeros for locations 0-4, and leading ones for locations 5-9. A simple cascade of AND and OR gates and a 1-bit multi-operand adder can be used to find the segment address for a given input δ_i as shown in Figure 4. The appropriate taps are taken from the cascades depending on the choice of the segments and are added to work out the P2S

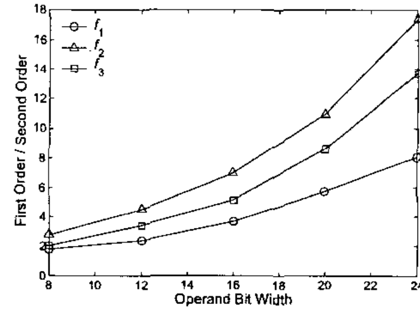

Figure 3. Ratio of the number of optimum segments required for first and second order approximations to the three functions.

Table 1. The ranges for P2S addresses for Λ_1 = P2S, $n = 8$, $v_0 = 5$ and $v_1 = 3$. The five P2S address bits δ_0 are highlighted in bold.

<table>
<tr><th>P2S address</th><th>range</th></tr>
<tr><td>0</td><td>**00000**|000~**00000**|111</td></tr>
<tr><td>1</td><td>**00001**|000~**00001**|111</td></tr>
<tr><td>2</td><td>**0001|0**000~**0001|1**111</td></tr>
<tr><td>3</td><td>**001|00**000~**001|11**111</td></tr>
<tr><td>4</td><td>**01|000**000~**01|111**111</td></tr>
<tr><td>5</td><td>**10|000**000~**10|111**111</td></tr>
<tr><td>6</td><td>**110|00**000~**110|11**111</td></tr>
<tr><td>7</td><td>**1110|0**000~**1110|1**111</td></tr>
<tr><td>8</td><td>**11110**|000~**11110**|111</td></tr>
<tr><td>9</td><td>**11111**|000~**11111**|111</td></tr>
</table>

address. For P2S that increase and decrease by powers of two, the full circuit is used, and for P2S that decrease only to the left side (P2SL), just the AND gates are used. Similarly for P2S that decrease to the right side (P2SR), the cascade OR gates are used. These circuits can be pipelined and a circuit with shorter critical path but requiring more area can be used [4]. Note that in the last partition, δ_λ is not used as an address. If Λ_i = US, then δ_{i+1} uses the next set of bits v_{i+1}. However if Λ_i = P2S, then the location of δ_{i+1} depends on the value of δ_i. Let j denote the P2S address, where $j = 0..s_i - 1$. From the vertical lines in Table 1, we observe that δ_{i+1} should be placed after a_0 for $j = 0$ and $j = s_i - 1$, after a_{j-1} for $j = 1$ to $j = (s_i/2) - 1$, and after a_{s_i-2-j} for $j = s_i/2$ to $j = s_i - 2$.

*In principle it is possible to have any number of lev*els of nested Λ, as long as $\sum_{i=0}^{\lambda} v_i \leq n$. The more lev-

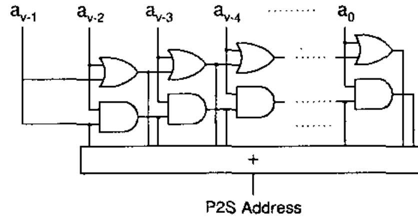

Figure 4. Circuit to calculate the P2S address for a given input δ_i, where $\delta_i = a_{v-1}a_{v-2}..a_0$. The adder counts the number of ones in the output of the two prefix circuits.

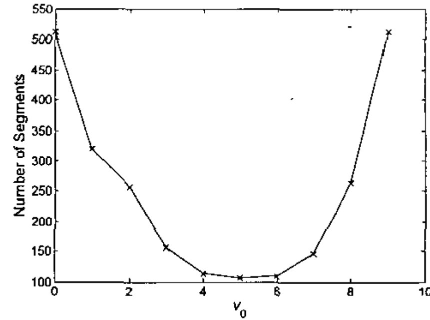

Figure 5. Variation of total number of segments against v_0 for a 16-bit second order approximation to f_3.

els are used, the closer the total number of segments m will be to the optimum. However as λ (the number of levels) increases the partitioning problem becomes more complex, and the cascade of look-up tables gets longer, increasing the delay to find the final segment. Therefore there is a tradeoff between the partitioning complexity, delay and m. Our tests with the functions we consider in this paper show that the rate of reduction of m decreases rapidly as λ increases. $\lambda = 2$ gives a very close m to the optimum with acceptable partitioning complexity and delay. Moreover, $\lambda > 2$ gives diminishing returns in terms of small improvement in m with high partitioning complexity and long delays. Therefore, in this paper we limit ourselves to $\lambda = 2$, which consists of one outer segment Λ_0 and one inner segment Λ_1. P2S is used as the outer segment if the function varies exponentially in the beginning and the end of the interval. P2SL and P2SR are used as the outer segment when the function varies exponentially at the beginning or at the end respectively. US is used if the function is non-linear in arbitrary regions. Although we limit ourselves with $\lambda = 2$, higher levels of hierarchies could be useful for certain functions. The hierarchy schemes we have chosen are $H = \{$P2S(US), P2SL(US), P2SR(US), US(US)$\}$. These four schemes cover most of the non-linear functions of interest.

We have implemented a hierarchical function segmenter (HFS) in MATLAB, which deals with the four schemes. The program takes as inputs the function f to be approximated, input range, operand size n, hierarchy scheme H, number of bits for the outer segment v_0, the requested output error e_{max}, and the precision of the polynomial coefficients and the data paths. HFS divides the input interval into outer segments whose boundaries are determined by H and v_0. HFS finds the optimum number of bits v_1 for the inner segments for each outer segment, which meets the requested output error constraint. For each outer segment, HFS starts with $v_1 = 0$ and computes the error e of the approximation. If $e > e_{max}$ then v_1 is incremented by one and the error e for each inner segment is computed, i.e. the number of inner segments is doubled in every iteration. If it detects that $e > e_{max}$ it increments v_1 again. This process is repeated until $e \leq e_{max}$ for all inner segments of the current outer segment. This is the point HFS obtains the optimum number of bits for the current outer segment. Once this process is performed for all outer segments, HFS generates the a look-up table containing the polynomial coefficients or each of the segments. It also generates a report, which contains the total number of segments m, maximum error, percentage of exactly rounded results, and the sizes of the multipliers, adders and look-up tables.

Experiments are carried out to find the optimum number of bits for the outer segment v_0. Figure 5 shows how the total number of segments varies with v_0 for 16-bit second order approximation to f_3. We can observe the figure of U shape, and there is a point at which v_0 is optimum, which is five bits in this particular case. When v_0 is too small, there are not enough outer segments to cater to local non-linearities. When v_0 is too large, there are too many unnecessary outer segments. Note that when $v_0 = 0$, it is equivalent to using standard uniform segmentation. Figure 6 shows the segmented functions obtained from HFS for 16-bit second order approximations to the three functions. It can be seen that the segments produced by HFS closely resemble the optimum segments in Figure 2. Table 2 shows a comparison in terms of numbers of segments for various second order approximations for uniform, HFS, and the optimum number of segments. Double precision is used for the data paths and the output for this comparison. We can see that HFS is significantly more efficient than using uniform segments, and the difference between the optimum ones are around a factor of 1.7. Looking at the results for 24-bit approximation to f_1, we can see that HFS performs

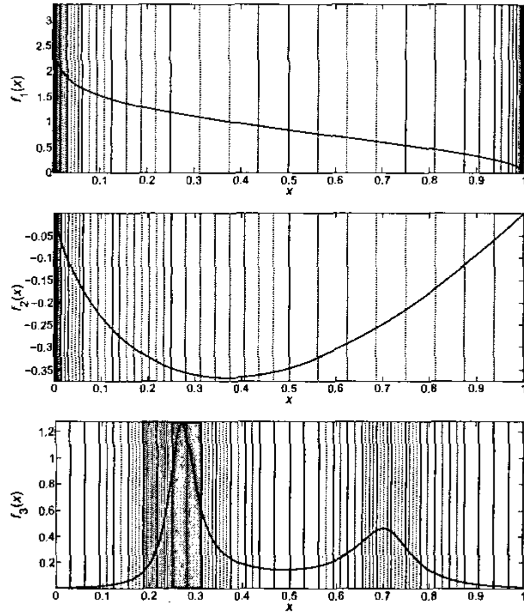

Figure 6. The segmented functions generated by HFS for 16-bit second order approximations. f_1, f_2 and f_3 employ P2S(US), P2SL(US) and US(US) respectively. The black and grey vertical lines are the boundaries for the outer and inner segments respectively.

worse than average. This is due to the fact that insufficient bits are left for δ_1 (19 bits are already used for δ_0).

An interesting aspect of our approach is that it could be used to accelerate applications that have involve pure floating point hardware such as software applications. This is because our method computes compound functions at once using polynomial approximations, instead of decomposing the compound functions into sub-functions and computing the sub-functions one by one.

5 Architecture

The architecture of our function evaluator for HFS is shown in Figure 7. The P2S unit performs the P2S address calculation (Figure 4) on δ_0 if δ_0 is of type P2S (Λ_0 = P2S). If Λ_0 = US, δ_0 is bypassed. The bit selection unit selects the appropriate bits from the input based on the values of v_0 and v_1. This variable bit selection is implemented using a barrel shifter. There are two look-up tables: one used for storing the v_1 values and the offset (ROM0), and the other storing the polynomial coefficients (ROM1). The offset in ROM0 stores the starting address in ROM1 for the different

Table 2. Number of segments for second order approximations to the three functions. Results for uniform, HFS and optimum are shown.

function	operand width	uniform segments	HFS segments	optimum segments	HFS /optimum
f_1	8	8	5	4	1.25
	12	1,024	23	15	1.53
	16	32,768	72	44	1.64
	20	524,288	218	126	1.73
	24	16,777,216	742	287	2.59
f_2	8	8	5	4	1.25
	12	128	15	10	1.50
	16	2,048	44	26	1.69
	20	32,768	124	66	1.88
	24	524,228	315	167	1.89
f_3	8	64	20	10	2.00
	12	256	41	24	1.71
	16	512	107	59	1.81
	20	1,024	234	151	1.55
	24	2,048	573	379	1.51

δ_0 values. The depth s_0 of ROM0 is defined in Equations (8) and (9), and the depth of m of ROM1 is the total number of segments. The size of the two look-up tables are defined as follows:

$$\text{ROM0} = \{\lceil \log_2(\max(v_1)) \rceil + \lceil \log_2(\max(\text{offset})) \rceil\} \times s_0 \quad (10)$$

$$\text{ROM1} = \sum_{i=0}^{d} w_i \times m \quad (11)$$

In practice, ROM0 is significantly smaller than ROM1, since the depth is bounded by v_0 and the entries v_1 and offset are small. There is an interesting tradeoff factor for ROM1. the wider the widths of the coefficients w, the fewer segments m are needed, since the approximations will be more accurate. However, if w is over a certain threshold, it has negligible effect on m. It is desirable to find the right widths that minimize the total ROM size.

Let b_j denote the the boundaries of the outer segments where $j = 0..m-1$ and $\theta = \max(b_j - b_{j+1})$ which is the maximum width of the outer segment. Instead of approximating each interval over $[b_i, b_{i+1})$, we perform the translation $\hat{x} = x - b_i$, which translates the interval $[b_i, b_{i+1})$ to $[0, \theta)$. This form reduces the widths of the data paths, since $\hat{x} \in [0, \theta)$ requires fewer bits to represent than x.

Highly non-linear functions such as f_1 which have exponentially varying regions to infinity, have a large dynamic range on the coefficients. For instance, the largest coefficient c_2 of a 24-bit second order approximation to f_1 is in the order of 10^{12}. In such cases, floating-point arithmetic is needed. For f_2 and f_3, where the ranges of the coefficients

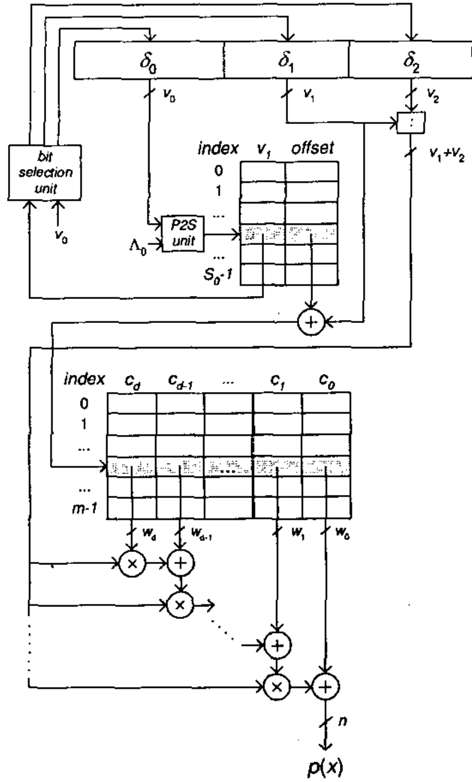

Figure 7. HFS function evaluator architecture for $\lambda = 2$ and degree d approximations. Note that ':' is a concatenation operator.

are relatively small, standard fixed point arithmetic is used.

For typical applications targeting FPGAs, the size of the two ROMs are small and can be implemented on-chip using distributed RAM or block RAM. Often the multiplier would be the part taking up a significant portion of the area. The size of the multipliers depend on the width of v_1+v_2 and the coefficients. Recent FPGAs, such as Xilinx Virtex-II or Altera Stratix devices, provide dedicated hardware resources for multiplication which can benefit the proposed architecture.

6 Evaluation and Results

Table 3 compares our approach with direct table look-up, SBTM and STAM for 16 and 24-bit second order approximations to f_2. We observe that table sizes for direct look-up approach are not feasible when the accuracy requirement is high. SBTM/STAM significantly reduce the table sizes compared to the direct table look-up approach, at the expense of some adders and control circuitry. However, we can see that the table size for HFS is 4 to 12 times smaller than SBTM/STAM, at the expense of two multipliers and three adders, hence higher latency. The difference between HFS and other methods gets larger as the bit width increases. HSF achieves smaller table size than SBTM/STAM at the expense of more complexity in terms of multipliers and adders, and the difference in table size gets larger as the accuracy requirement increases. For applications that require relatively low accuracies and low latencies, SBTM/STAM may be preferred. For high accuracy applications that can tolerate more latencies, HFS would be more appropriate.

Table 3. Comparison of direct look-up, SBTM, STAM and HFS for 16 and 24-bit second order approximation to f_2. The subscript for STAM denotes the number of tables used. SBTM is equivalent to $STAM_2$.

operand width	method	table size (bits)	compression	multiplier	adder
16	direct	1,048,576	227	-	-
	SBTM	29,696	6	-	1
	$STAM_4$	16,384	4	-	3
	HFS	4,620	1	2	3
24	direct	402,653,184	9,955	-	-
	SBTM	2,293,760	57	-	1
	$STAM_6$	491,520	12	-	5
	HFS	40,446	1	2	3

A variant [7] of our approximation scheme to f_1 and trigonometric functions, with one level of P2S and US(P2S), has been implemented and successfully used for the generation of Gaussian noise samples [6]. Table 4 contains results for f_2 and f_3 with 16 and 24-bit operands and second order approximation. The designs have been implemented with Xilinx System Generator, and are mapped and tested on a Xilinx Virtex-II XC2V4000-6 FPGA. The precision of bit width and the data paths have been optimized to minimize the size of the multipliers and look-up tables. The design is fully pipelined generating a result every clock cycle. Designs with lower latency and clock speed can be obtained by reducing the number of pipeline stages. The designs have been tested exhaustively over all possible input values to verify that all outputs are indeed faithfully rounded. Although we have not synthesized the designs for SBTM and STAM, we estimate that they will take significantly less area in terms of slices than HFS (since only

Table 4. Hardware synthesis results on a Xilinx Virtex-II XC2V4000-6 FPGA for 16 and 24-bit second order approximations to f_2 and f_3.

function	operand width	speed (MHz)	latency (cycles)	slices	block RAMs	block multipliers
f_2	16	153	13	483	1	4
	24	135	14	871	2	10
f_3	16	198	12	234	1	3
	24	157	13	409	3	4

adders and some control circuity are required, and adders are efficiently implemented on Xilinx FPGAs using fast carry chains), but at the expense of more block RAM usage. The difference of block RAM usage between HFS and SBTM/STAM will get more significant as the accuracy requirement increases as shown in Table 3.

7 Conclusion

We have presented a novel method for evaluating functions using piecewise polynomial approximations by with an efficient hierarchical segmentation scheme. An algorithm that finds the optimum segments for a given function, input range, maximum error and ulp has been presented. The four hierarchical schemes P2S(US), P2SL(US), P2SR(US) and US(US) 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 P2S address for a given input. Results show the advantages of using our hierarchical approach over the traditional uniform approach. Compared to other popular methods, our approach has longer latency and more operators, but the size of the look-up tables are considerably smaller. Current and future work includes extending the hierarchical function segmenter (HFS) to cover functions with two variables, floating point arithmetic implementation and automating the optimization of the widths of coefficients and data paths. We will also look at how HFS can be used to speed up addition and subtraction functions in logarithmic number systems, which are highly non-linear.

Acknowledgment

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

References

[1] D. Das Sarma and D.W. Matula, "Faithful bipartite rom reciprocal tables", *Proc. 12th IEEE Symp. on Computer Arithmetic*, pp. 17–28, 1995.

[2] R.E. Esch and W.L. Eastman, "Computational methods for best spline approximation", *J. of Approx. Theory*, vol. 2, pp. 85–96, 1969.

[3] C. Jones, E. Vallés, M. Smith and J. Villasenor, "A high throughput low complexity decoder architecture for irregular LDPC codes", *Proc. IEEE Military Comm. Conf. (MILCOM)*, 2003.

[4] R.E. Ladner and M.J. Fischer, "Parallel prefix computation", *J. of the ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[5] C.L. Lawson, "Characteristic properties of the segmented rational minimax approximation problem", *Numer. Math.*, vol. 6, pp. 293–301, 1964.

[6] 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.*, pp. 69–78, 2003.

[7] D. Lee, W. Luk, J. Villasenor and P.Y.K. Cheung, "Hardware function evaluation using non-linear segments", *Proc. Field-Prog. Logic and Applications*, LNCS 2778, Springer-Verlag, pp. 796–807, 2003.

[8] J.M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser Verlag AG, 1997.

[9] T. Pavlidis, "Waveform segmentation through functional approximation", *IEEE Trans. on Comput.*, vol. C-22, no. 7, pp. 689–697, 1973.

[10] J.R. Rice, *The Approximation of Functions*, vol. 1,2, Addison-Wesley, 1964, 1969.

[11] D. Rueckert, L.I Sonoda, C. Hayes, D.L. Hill, M.O. Leach and D.J. Hawkes, "Nonrigid registration using free-form deformations: application to breast MR images", *IEEE Trans. on Medical Imaging*, vol. 18, no. 8, pp. 712–720, 1999.

[12] 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.

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