Hardware Implementation Trade-Offs of Polynomial Approximations and Interpolations

Dong-U Lee, *Member*, *IEEE*, Ray C.C. Cheung, *Member*, *IEEE*,
Wayne Luk, *Senior Member*, *IEEE*, and John D. Villasenor, *Senior Member*, *IEEE*

Abstract—This paper examines the hardware implementation trade-offs when evaluating functions via piecewise polynomial approximations and interpolations for precisions of up to 24 bits. In polynomial approximations, polynomials are evaluated using stored coefficients. Polynomial interpolations, however, require the coefficients to be computed on-the-fly by using stored function values. Although it is known that interpolations require less memory than approximations, but at the expense of additional computations, the trade-offs in memory, area, delay, and power consumption between the two approaches have not been examined in detail. This work quantitatively analyzes these trade-offs for optimized approximations and interpolations across different functions and target precisions. Hardware architectures for degree-1 and degree-2 approximations and interpolations are described. The results show that the extent of memory savings realized by using interpolation is significantly lower than what is commonly believed. Furthermore, experimental results on a field-programmable gate array (FPGA) show that, for high output precision, degree-1 interpolations offer considerable area and power savings over degree-1 approximations, but similar savings are not realized when degree-2 interpolations and approximations are compared. The availability of both interpolation-based and approximation-based designs offers a richer set of design trade-offs than what is available using either interpolation or approximation alone.

Index Terms—Algorithms implemented in hardware, interpolation, approximation, VLSI systems.

1 Introduction

The evaluation of functions is essential to numerous signal processing, computer graphics, and scientific computing applications, including direct digital frequency synthesizers [1], Phong shaders [2], geometrical transformations [3], and N-body simulations [4]. Dedicated hardware-based function evaluation units on field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs) are often desired over their software-based counterparts due to their huge speed advantages.

Direct lookup tables are sometimes used due to their ease of design and fast execution times. However, the table size grows exponentially with the number of bits at the input and can become impractically large for high input precisions. Iterative techniques such as CORDIC [5] have been popular, but they are less suitable for high throughput applications due to their multicycle execution delays. Function approximation via weighted sum of bit products was recently proposed, which was shown to lead to improved throughput and area characteristics over CORDIC [6]. Polynomial-only approximations have the advantage of being ROM-less, but they can impose large computational complexities and delays [7]. Table addition methods [8] provide a good balance between computation and memory, without the need for multipliers, but their memory requirements can become large for precisions beyond 16 bits.

Our research covers table-based methods using piecewise polynomials, which are generally considered to be suitable for low-precision arithmetic of up to 32 bits. Furthermore, they offer flexible design trade-offs involving computation, memory, and precision. The input interval is partitioned into multiple segments and a (typically) low-degree polynomial is used to evaluate each segment. The evaluation accuracy can be controlled by varying the number of segments and/or the polynomial degree. With piecewise polynomials, one can opt for either "approximation" or "interpolation." Approximation is "the evaluation of a function with simpler functions," whereas interpolation is "the evaluation of a function from certain known values of the function" [9]. Hence, in this paper, in piecewise polynomial approximations, each segment is associated with a set of table entries, giving the coefficients for the appropriate approximating polynomial. In contrast, in piecewise polynomial interpolation, the function values at the segment end points are stored and the coefficients of approximating polynomials are computed at runtime [10]. Thus, in a broad sense, interpolations can be regarded as approximations as well, but, as is customary in the literature, we shall use the terms "approximation" and "interpolation" to distinguish the first and second approaches described above.

Both methods have their advantages and disadvantages. To achieve a given precision, interpolations potentially require smaller tables than approximations since a single

- *D. Lee is with Mojix, Inc., 11075 Santa Monica Blvd., Suite 350, Los Angeles, CA 90025. E-mail: dongu@mojix.com.*
- *R.C.C. Cheung is with Solomon Systech Limited, No. 3 Science Park East Avenue, Hong Kong Science Park, Hong Kong. E-mail: cccheung@ieee.org.*
- *W. Luk is with the Department of Computing, Imperial College London, London, UK. E-mail: w.luk@imperial.ac.uk.*
- *J.D. Villasenor is with the Electrical Engineering Department, University of California, Los Angeles, 420 Westwood Blvd., Los Angeles, CA 90095-1594. E-mail: villa@icsl.ucla.edu.*

Manuscript received 20 Dec. 2006; revised 27 July 2007; accepted 23 Oct. 2007; published online 29 Oct. 2007.
Recommended for acceptance by M. Gokhale.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0478-1206.
Digital Object Identifier no. 10.1109/TC.2007.70847.

0018-9340/08/$25.00 © 2008 IEEE Published by the IEEE Computer Society

function value, rather than a set of coefficients, is stored for each segment. However, interpolations impose higher computational burdens due to the coefficient computation step. Although there is a significant body of literature addressing implementation methods and associated trade-offs, either for approximation or interpolation alone, the detailed memory, area, delay, and power trade-offs *between* the two methods have not been investigated.

In this paper, we quantitatively examine such trade-offs for degree-1 and degree-2 designs across various functions, segmentation methods, and target precisions. Among the more notable findings are that, for degree-2 designs, the average memory savings obtainable by using interpolation, instead of approximation, is under 10 percent, which is significantly lower than the commonly believed savings of 20 percent to 40 percent [11], [12]. This is primarily due to the effect of the memory location bit widths, which was not taken into account in previous work. Another interesting result is that, for degree-1 interpolations, the increase in circuit area due to the additional computations is more than compensated for by the decrease in area due to lower memory requirements. This leads to significant area and power advantages over degree-1 approximations.

To summarize, the main contributions of this paper are:

- propose a common framework for capturing the design flow and error analysis of both approximation and interpolation designs,
- examine the application of uniform and hierarchical segmentations,
- review and compare hardware architectures for both approximation and interpolation methods,
- apply techniques based on analytical bit width optimization and resource estimation for improving speed, area, and power consumption, and
- present experimental results targeting Xilinx FPGAs to illustrate and evaluate our approach.

In what follows, precision is quantified in terms of the unit in the last place (ulp). The ulp of a fixed-point number with n fractional bits (FBs) would be 2^{-n}. We target "faithful" rounding in which results are rounded to either the nearest or the next nearest fraction expressible using the available bits and are thus accurate to within 1 ulp.

This paper focuses on degree-1 and degree-2 designs for the following reasons: First, degree-1 and degree-2 polynomials are generally regarded as the most efficient for the target precisions of 10 to 24 bits considered in this work [13]. Second, the evaluation of coefficients from function values for degree-1 and degree-2 interpolations involves multiplications that are powers of two, thus significantly reducing the implementation cost. That said, higher degree approximations are commonly used in situations where memory requirements must be minimized at the expense of increased computation or when precisions beyond 24 bits are required [13], [14].

2 Related Work

Between the two methods, polynomial approximation has received more attention in the literature. Noetzel [15] examined piecewise polynomial approximations involving uniformly sized segments with Lagrange coefficients. It was demonstrated that, by adjusting the polynomial degree for a given target precision, the function can be approximated with a variety of trade-offs involving computation and memory. Takagi [16] presented a degree-1 approximation architecture for performing powering operations. The multiplication and addition involved in degree-1 approximation were replaced with a larger multiplication and operand modification. Single multiplication degree-2 architectures were proposed in [17], [18]. A multiplier was eliminated through precomputing partial polynomial terms at the expense of higher memory requirements. Piñeiro et al. [19] proposed a highly optimized degree-2 architecture with a dedicated squarer and a fused accumulation tree. Their implementations result in significant reductions in table size, with a slight increase in execution time compared to other methods. Lee et al. [13] explored the design space of different-degree piecewise approximations in terms of area, latency, and throughput on FPGAs. It was demonstrated that polynomials with certain degrees were better than others for a given metric and target precision. Schulte and Swartzlander [20] studied the impact of achieving an exact rounding (1/2 ulp of accuracy) on the area and delay with polynomial approximations. Their results indicated that the exact rounding typically imposed 33 percent to 77 percent of hardware area penalty over the faithful rounding. Walters and Schulte [21] described degree-1 and degree-2 architectures with truncated multipliers and squarers. Their approach required up to 31 percent fewer partial product computations compared to approximations with standard multipliers/squarers.

One of the earliest examinations of digital interpolation was performed by Aus and Korn [22] in the 1960s, who examined software routines for degree-1 interpolations for sine and cosine functions on the DEC PDP-9 platform. Lewis [11] described an interleaved memory architecture for interpolation and its application to evaluating the addition/subtraction functions in logarithmic number systems. It was estimated that, compared to approximations, interpolations used 30 percent to 50 percent less memory for degree-1 designs and 20 percent to 40 percent less memory for degree-2 designs. Cao et al. [12] examined degree-2 interpolation circuits for the evaluation of elementary functions. Several variants of degree-2 interpolation that trade off computation and memory were investigated. Cao et al. state that degree-2 interpolations use 33 percent less memory than approximations, a result that is consistent with the range provided in [11]. Paliouras et al. [23] explored degree-2 interpolation hardware for evaluating sine and cosine functions. The interval was partitioned nonuniformly to minimize the number of function values required. McCollum et al. [24] employed degree-1 interpolations for the evaluation of the inverse Gaussian cumulative distribution functions. Lamarche and Savaria [25] studied the mapping of degree-1 interpolations on FPGAs. Synthesis results for the interpolation of the error function on a Xilinx Virtex XCV300 FPGA were presented. As noted earlier, the contributions cited above address approximations and interpolations separately, whereas this

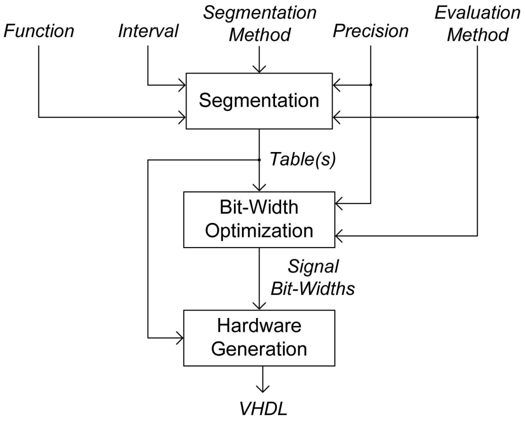


Fig. 1. Design flow for polynomial approximation and interpolation hardware design.

present work investigates the hardware implementation trade-offs of the two approaches.

3 Framework

Consider an elementary function $f(x)$, where x is to be evaluated over a range $[a, b]$ and to a given target precision requirement. The evaluation of $f(x)$ typically consists of the following steps [7]:

1. range reduction: reducing the input interval $[a, b]$ to a smaller interval $[a', b']$;
2. function approximation/interpolation over the reduced interval;
3. range reconstruction: expanding the result back to the original result range.

Since range reductions and reconstructions are well-studied topics, we focus on the approximation/interpolation of a function over the reduced interval.

Fig. 1 depicts the design flow for polynomial approximation and interpolation hardware design. The following input parameters are required:

1. target function (for example, $\ln(1 + x)$),
2. evaluation interval (for example, $x = [0, 1)$),
3. segmentation method (uniform or nonuniform),
4. target output precision (for example, 20 FBs), and
5. evaluation method (approximation or interpolation) and degree of polynomials (for example, degree-2 interpolation).

The target function can be any continuous differentiable function, including elementary functions and compound functions. Arbitrary evaluation intervals of interest can be specified. A "segment" refers to the subinterval over which a set of precomputed coefficients are used for the case of approximation and for which the starting and ending function values are stored for the case of interpolation. The two segmentation options are 1) uniform segmentation, in which the segment widths are equal, and 2) nonuniform segmentation, in which the segment widths can be variable. The desired target precision is specified in terms of the number of FBs. Since faithful rounding is used, specifying 20 FBs, for instance, would lead to a worst-case error bound of less than or equal to 2^{-20} at the output.

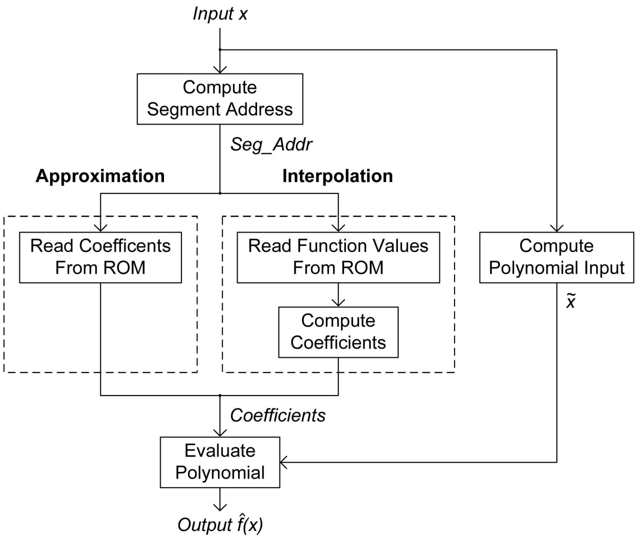


Fig. 2. Overview of the steps involved in polynomial approximation and interpolation.

The first step of the design flow in Fig. 1 is segmentation. For a given segmentation method, this step finds the minimal number of segments while respecting the error constraint of the target precision. Once segmentation is completed, a table containing the polynomial coefficients (in case of approximation) or the set of function values (in case of interpolation) is generated. In addition, if nonuniform segmentation is selected, an additional table holding the segmentation information is also produced. The second step, that is, bit width optimization, identifies the required bit width for each fixed-point operand in the data path. The last step is hardware generation, which uses the table(s) and the operand bit widths to generate synthesizable VHDL code. In this flow, certain portions of the total error budget are preallocated to the segmentation step (for inherent approximation/interpolation errors) and the bit width optimization step (for finite-precision effects). This avoids the need to include feedback from the hardware generation step to the segmentation step, which would greatly complicate the design process with little or no benefit to the resulting design.

Fig. 2 shows an overview of the computational steps involved in polynomial approximation and interpolation. Given the input x, its corresponding segment address *Seg_Addr* and the input argument $\tilde{x}$ for the polynomial evaluation is computed. $\tilde{x}$ is given by $\tilde{x} = (x - x_i)/h$, where x_i is the x-coordinate of the beginning of the current segment, and h is the segment width. For approximation, *Seg_Addr* simply serves as the index to the polynomial coefficient ROM. For interpolation, *Seg_Addr* indexes the ROM(s) from which the function values need to be fetched and the polynomial coefficients are then computed on-the-fly from the function values. In both methods, polynomial evaluation is performed via the coefficients and the polynomial input $\tilde{x}$ to produce the approximated/interpolated output $\hat{f}(x)$. The resulting approximation/interpolation designs can then be implemented in a variety of

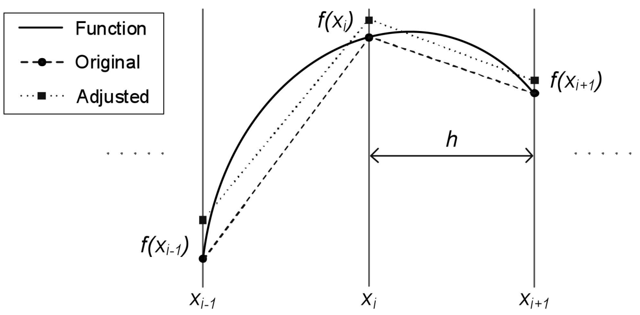


Fig. 3. Illustration of degree-1 interpolation.

technologies. Section 7 covers speed, area, and power consumption variations for the FPGA technology.

3.1 Polynomial Approximation

Approximation involves the evaluation of functions via simpler functions. In this work, the simpler functions are realized via piecewise polynomials. Different types of polynomial approximations exist with respect to the error objective, including least square approximations, which minimize the root mean square error, and least maximum approximations, which minimize the maximum absolute error [7]. When considering designs that meet constraints on the maximum error, least maximum approximations are of interest. The most commonly used least maximum approximations include the Chebyshev and minimax polynomials. The Chebyshev polynomials provide approximations close to the optimal least maximum approximation and can be constructed analytically. Minimax polynomials provide slightly better approximations than Chebyshev but must be computed iteratively via the Remez algorithm [26]. Since minimax polynomials have been widely studied in the literature (for example [13], [18], [19]) and offer better approximation performance, they are adopted for our work here. The Horner rule is employed for the evaluation of polynomials in the following form:

$$\hat{f}(x) = ((C_d\tilde{x} + C_{d-1})\tilde{x} + \ldots)\tilde{x} + C_0, \quad (1)$$

where $\tilde{x}$ is the polynomial input, d is the degree, and $C_{0..d}$ are the polynomial coefficients.

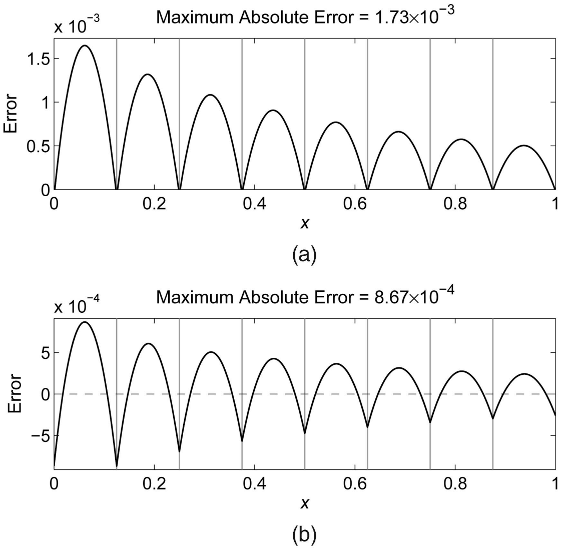
3.2 Polynomial Interpolation

Interpolation is a method of constructing new data points from a discrete set of known data points. In contrast with polynomial approximation-based hardware function evaluation, interpolation has received somewhat less attention in the research community. In this section, we address degree-1 and degree-2 interpolations.

3.2.1 Degree-1 Interpolation

Degree-1 interpolation uses a straight line that passes through two known points $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$, where $x_i < x_{i+1}$, as illustrated by the dashed line in Fig. 3. At a point $x = [x_i, x_{i+1})$, the point-slope formula can be used for the interpolation of $\hat{f}(x)$:

$$\hat{f}(x) = (f(x_{i+1}) - f(x_i))\frac{x - x_i}{x_{i+1} - x_i} + f(x_i). \quad (2)$$



(a)

(b)

Fig. 4. Interpolation error when evaluating $\ln(1+x)$ over $x = [0, 1)$ with degree-1 interpolations using nine equally spaced function values. (a) Original. (b) Adjusted.

The standard degree-1 polynomial is given by $\hat{f}(x) = C_1\tilde{x} + C_0$. Examining (2), we find that

$$C_1 = f(x_{i+1}) - f(x_i), \quad (3)$$

$$C_0 = f(x_i), \quad (4)$$

$$\tilde{x} = \frac{x - x_i}{h}, \quad (5)$$

where $h = x_{i+1} - x_i$. The computation of the coefficients requires two lookup tables and one subtraction. The worst-case approximation error is bounded by [27]

$$\epsilon_{\max} = \frac{h^2}{8}\max(|f''(x)|), \quad \text{where } x = [x_i, x_{i+1}). \quad (6)$$

Fig. 4a shows the error plot when evaluating $\ln(1+x)$ over $x = [0, 1)$ with degree-1 interpolations using nine equally spaced function value points corresponding to eight segments. The average interpolation error generally decreases with an increasing x because the magnitude of the second derivative of $\ln(1+x)$ decreases with x (6). The maximum error occurs in the interpolation between the first two function values over $x = [0, 0.125)$ (first segment). Using (6), this error is bounded at 1.95×10^{-3}. However, (6) is a loose bound and, in fact, the actual interpolation error is 1.73×10^{-3}. The knowledge of the exact interpolation error is essential for producing optimized hardware, as will be discussed in Section 6. In order to compute the exact interpolation error for each segment, we first find the root of the derivative of $f(x) - \hat{f}(x)$, where $f(x)$ is the true function and $\hat{f}(x)$ is the interpolating polynomial of the segment. The root is the x value at which the maximum interpolation error occurs. Finally, this x value is substituted into $f(x) - \hat{f}(x)$ to give the exact interpolation error of the segment.

In Fig. 4a, it is observed that the errors have the same sign throughout the interval. This will always be the case for functions whose derivative is monotonically increasing or decreasing over the interpolation interval. As noted by Lewis [11], it is possible to reduce the errors in such situations by adjusting the function values. The specific-adjustment method [11] assumes that the maximum error occurs precisely at the midpoint of the segment. However, although the maximum error position is typically near the midpoint, it is not typically exactly at the midpoint. By using an alternate adjustment methodology described as follows, a slight improvement in the postadjustment error can be obtained:

Let seg_err_j denote the maximum interpolation error of the jth segment and M denote the number of segments. The first and last function values $f(x_0)$ and $f(x_M)$ are reduced by $seg\_err_0/2$ and $seg\_err_{M-1}/2$, respectively. Each intermediate function value $f(x_i)$, however, affects seg_err_{i-1} and seg_err_i. To balance the errors of the consecutive segments, $(seg\_err_{i-1} + seg\_err_i)/4$ is subtracted from the intermediate function values. Fig. 4b shows the error plot of the adjusted function values. Unlike the error plot of the original function values (Fig. 4a), the error behaves in a symmetric manner around $f(x) = 0$, reducing the maximum absolute error by a factor of two to 8.67×10^{-4}. The adjusted function values are illustrated in Fig. 3. For segments on the boundaries, each of such segments has a function value not shared by any other segments. This adjustment process leads to a degree-1 interpolating line that has the same maximum-error properties as the minimax degree-1 approximation. For interior segments, however, the degree-1 interpolation and degree-1 approximations will differ.

3.2.2 Degree-2 Interpolation

The degree-2 Lagrange interpolating polynomial through the three points $(x_{i-1}, f(x_{i-1}))$, $(x_i, f(x_i))$, and $(x_{i+1}, f(x_{i+1}))$, where $x_{i-1} < x_i < x_{i+1}$, is [27]

$$\begin{aligned}\hat{f}(x) = {} & f(x_{i-1})\frac{(x - x_i)(x - x_{i+1})}{(x_{i-1} - x_i)(x_{i-1} - x_{i+1})} \\ & + f(x_i)\frac{(x - x_{i-1})(x - x_{i+1})}{(x_i - x_{i-1})(x_i - x_{i+1})} \\ & + f(x_{i+1})\frac{(x - x_{i-1})(x - x_i)}{(x_{i+1} - x_{i-1})(x_{i+1} - x_i)}.\end{aligned} \tag{7}$$

Assuming that the function values are equally spaced, substituting $h = x_i - x_{i-1} = x_{i+1} - x_i$ and $\tilde{x} = (x - x_i)/h$ into (7) gives

$$\begin{aligned}\hat{f}(x) = {} & \left(\frac{f(x_{i+1}) + f(x_{i-1})}{2} - f(x_i)\right)\tilde{x}^2 \\ & + \frac{f(x_{i+1}) - f(x_{i-1})}{2}\tilde{x} + f(x_i).\end{aligned} \tag{8}$$

Since the degree-2 polynomial in the Horner form is given by $\hat{f}(x) = (C_2\tilde{x} + C_1)\tilde{x} + C_0$, from (8), the coefficients are given by

$$C_2 = \frac{f(x_{i+1}) + f(x_{i-1})}{2} - f(x_i), \tag{9}$$

$$C_1 = \frac{f(x_{i+1}) - f(x_{i-1})}{2}, \tag{10}$$

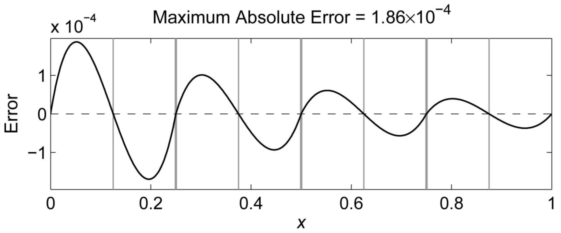


Fig. 5. Interpolation error when evaluating $\ln(1 + x)$ over $x = [0, 1)$ with degree-2 method 1 interpolations using nine equally spaced function values.

$$C_0 = f(x_i), \tag{11}$$

which require three lookup tables, three additions and subtractions, and constant shifts. The worst-case approximation error is bounded by [27]

$$\epsilon_{\max} = \frac{h^3}{9\sqrt{3}}\max(|f'''(x)|), \quad \text{where } x = [x_{i-1}, x_{i+1}). \tag{12}$$

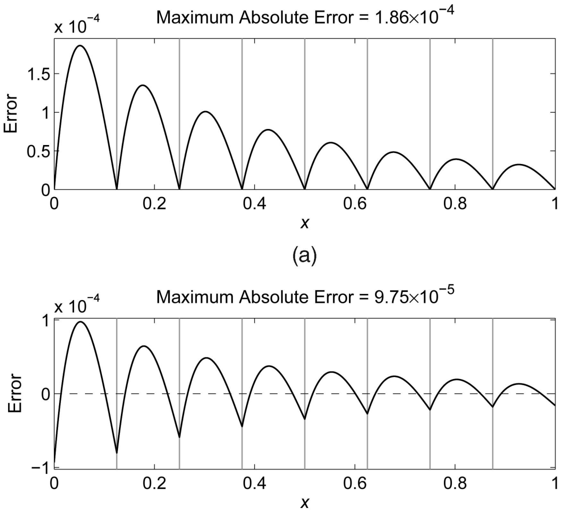
In degree-2 interpolation, the following strategies are possible:

- *Method 1.* Use $f(x_{i-1})$, $f(x_i)$, and $f(x_{i+1})$ for the interpolation over $x = [x_{i-1}, x_{i+1})$.
- *Method 2.* Use $f(x_{i-1})$, $f(x_i)$, and $f(x_{i+1})$ for the interpolation over $x = [x_{i-1}, x_i)$ or $x = [x_i, x_{i+1})$ only.

Method 1 is employed by Paliouras et al. [23] and Cao et al. [12], whereas method 2 is employed by Lewis [11]. Although method 1 is simpler to implement from the hardware perspective (Section 5), method 2 can result in lower interpolation errors and allows the function values to be adjusted as discussed in Section 3.2.1 to reduce the maximum absolute error. For instance, consider the interpolation of a function over the range $x = [x_i, x_{i+1})$ with a decreasing third derivative. With method 1, function values $f(x_{i-1})$, $f(x_i)$, and $f(x_{i+1})$ will be used and the error will be bounded by $h^3/9\sqrt{3}|f'''(x_{i-1})|$ and its sign alternates. With method 2, however, function values $f(x_i)$, $f(x_{i+1})$, and $f(x_{i+2})$ will be used, resulting in a reduced error bound of $h^3/9\sqrt{3}|f'''(x_i)|$ and the error has constant sign, making it suitable for function value adjustments. Although method 2 lowers the interpolation error, it requires an extra function value. As will be discussed in Section 4, it imposes higher hardware complexity.

Fig. 5 shows the interpolation error when evaluating $\ln(1 + x)$ over $x = [0, 1)$ with degree-2 method-1 interpolations using nine equally spaced function values. To determine the interpolation error, instead of using the bound in (12), we use the exact error computation technique discussed in Section 3.2.1. Method 1 results in a maximum absolute error of 1.86×10^{-3} and the sign of the error alternates between successive segments.

Fig. 6a shows the same error plot when method 2 is used. For this particular example, the maximum absolute error is identical to method 1 since the error is dominated by the first segment and the set of function values used for the first segment is identical for both methods. However, the sign of the error in method 2 is constant throughout the interval, allowing the function value adjustment method described in Section 3.2.1 to be applied. The error plot of method 2 with adjusted function values is shown in Fig. 6b. Note that



(a)

(b)

Fig. 6. Interpolation error when evaluating $\ln(1+x)$ over $x = [0,1)$ with degree-2 method 2 interpolations using 10 equally spaced function values. (a) Original. (b) Adjusted.

an extra function value is required for interpolating the last segment. The interpolation error is 9.75×10^{-5}, which is about half the unadjusted interpolation error and an order of magnitude lower than the adjusted degree-1 interpolation error (Fig. 4b). When degree-2 minimax approximations are applied to the example above, the maximum absolute error is 1.70×10^{-5}, which is considerably lower than any of the degree-2 interpolation methods discussed above. In general, the degree-2 polynomials obtained through interpolation (whether adjusted or not) can deviate significantly from the optimal minimax polynomials. As noted earlier, in the case of degree 1, the differences between the interpolation polynomial (that is, a straight line) and the minimax line are much smaller.

4 Segmentation

The most common segmentation approach is uniform segmentation, in which all segment widths are equal, with the number of segments typically limited to powers of two. This leads to a simple and fast segment indexing. However, a uniform segmentation does not allow the segment widths to be customized according to local function characteristics. This constraint can impose high memory requirements for nonlinear functions whose first-order or higher order derivatives have high absolute values since a large number of segments are required to meet a given error requirement [28].

A more sophisticated approach is to use nonuniform segmentation, which allows the segment widths to vary. Allowing completely unconstrained segment widths would lead to a number of practical issues. Thus, we use hierarchical segmentation [28], which utilizes a two-level hierarchy consisting of an outer segmentation and an inner segmentation. The outer segmentation employs uniform segments or segments whose sizes vary by powers of two, whereas the inner segmentation always uses uniform segmentation. Compared to uniform segmentation, hierarchical segmentation requires significantly fewer segments for highly nonlinear functions such as the entropy computation $-x\ln(x)$ over $x = [0,1)$. For relatively linear functions such as $\ln(1+x)$ over $x = [0,1)$, moderate savings can still be achieved due to the constraint of uniform segmentation, in which the total number of segments must be a power of two. However, due to its nonuniformity, hierarchical segmentation requires extra circuitry for segment indexing.

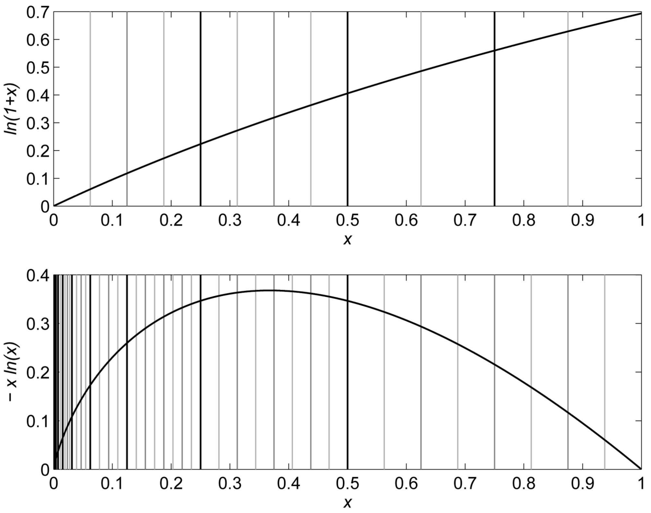


Fig. 7. Hierarchical segmentations to $\ln(1+x)$ and $-x\ln(x)$ using degree-2 method 1 interpolations at an error requirement of 2^{-14}. $\ln(1+x)$ requires four outer segments, resulting in a total of 12 segments, whereas $-x\ln(x)$ requires 10 outer segments, resulting in a total of 48 segments. The black and gray vertical lines indicate the boundaries for the outer segmentation and inner segmentation, respectively.

Fig. 7 illustrates the hierarchical segmentations for functions $\ln(1+x)$ and $-x\ln(x)$ using degree-2 method-1 interpolations for an error requirement of 2^{-14}. Uniform segments are used for the outer segmentation of $\ln(1+x)$, whereas segments that increase by powers of two are used for the outer segmentation of $-x\ln(x)$. $\ln(1+x)$ and $-x\ln(x)$ require a total of 12 and 48 segments, respectively. $\ln(1+x)$ uses four outer segments, whereas $-x\ln(x)$ uses 10 outer segments. Note that the number of uniform segments within each outer segment is variable. The figure demonstrates that the segment widths adapt to the nonlinear characteristics of the functions. If uniform segmentations are used instead, $\ln(1+x)$ requires 16 segments and $-x\ln(x)$ requires 2,048 segments.

Table 1 compares the number of segments and the number of function values that need to be stored for degree-1 and degree-2 interpolations with hierarchical segmentation. Degree-2 method-2 results use adjusted function values. Both degree-1 methods and degree-2 method 1 require storage of $M+1$ function values, where M is the number of segments, as noted in Section 3.2. However, an additional function value is required just before or after each outer segment for degree-2 method 2. The table shows that, as expected, the number of segments increases with the error requirement and degree-2 interpolations require significantly fewer segments than degree-1 interpolations. In degree-1 interpolations, adjusted function values reduce the required number of function values by up to 30 percent. For degree 2, little reduction is obtained by adopting method 2 over method 1. This is due to the overhead of the extra function value for each outer segment.

TABLE 1
Comparisons of the Number of Segments and Function Values for Interpolations with Hierarchical Segmentation

Function	Degree	Method	ϵ_{REQ}	Segments	Function Values
$\ln(1+x)$	1	Original	2^{-8}	6	7
			2^{-16}	91	92
		Adjusted	2^{-8}	4	5
			2^{-16}	64	65
	2	Method 1	2^{-8}	4	5
			2^{-16}	20	21
		Method 2	2^{-8}	4	6
			2^{-16}	16	18
$-x\ln(x)$	1	Original	2^{-8}	16	17
			2^{-16}	256	257
		Adjusted	2^{-8}	12	13
			2^{-16}	192	193
	2	Method 1	2^{-8}	12	13
			2^{-16}	80	81
		Method 2	2^{-8}	10	14
			2^{-16}	64	75

Degree-2 method 2 results use adjusted function values. ϵ_{REQ} refers to the error requirement.

Similar experiments are conducted for uniform segmentations. Results indicate that the degree-1 adjusted method and degree-2 method 2 can occasionally reduce the number of function values by half compared to the degree-1 original method and degree-2 method 1, respectively. In most cases though, the number of function values is identical. This is due to the fact that, with uniform segmentations, the number of segments always varies by powers of two. Hence, in certain boundary cases, a slight reduction in error can avoid the jump to the next power of two.

5 Hardware Architectures

As illustrated in Fig. 2 in Section 3, the first step is to compute the segment address *Seg_Addr* for a given input x. Let B_z denote the bit width of an operand z. With a uniform segmentation, segment address computation is nearly free since the leading $B_{x_{addr}}$ bits of the input x are simply used to address $2^{B_{x_{addr}}}$ segments. A hierarchical segmentation has the benefit of adaptive segment widths but at the price of extra hardware for segment address computation. If uniform segments are used for an outer segmentation (for example, $\ln(1+x)$ in Fig. 7), a barrel shifter is needed. If segments that vary by powers of two are selected for outer segmentation (for example, $-x\ln(x)$ in Fig. 7), a leading zero detector and two barrel shifters are required. In both cases, a small ROM for storing the segmentation information is necessary [28].

With approximations, the segment address is used to index the coefficient ROM. The coefficient ROM has M rows, where M is the number of segments. As illustrated in Fig. 9, each row is a concatenation of the polynomial coefficients to each segment.

With interpolations, the segment address indexes the ROMs that hold the function values. Fig. 8 shows degree-1 and degree-2 method 1 single-port ROM architectures for extracting the corresponding function values to each segment. These architectures work for both uniform and hierarchical segmentations. The degree-1 single-port design in Fig. 8a uses ROM0 and ROM1 as the interleaved memories. As illustrated in Fig. 10, ROM0 stores function values with even indices, whereas ROM1 stores function values with odd indices. The incrementer and the two shifters ensure that the correct function values are extracted. The least significant bit (LSB) of *Seg_Addr* is used as the select signal of the two multiplexers to correctly order the two function values read from the ROMs. The degree-2 single-port design in Fig. 8b works in the same way as the degree-1 case, except that it has an extra ROM (ROM2). ROM2 is used to store the midpoint $f(x_i)$ between the function values $f(x_{i-1})$ and $f(x_{i+1})$ of ROM0 and ROM1. However, due to this midpoint approach, *Seg_Addr* is used to address two consecutive segments.

Fig. 11 depicts the corresponding degree-1 and degree-2 architectures utilizing dual-point ROMs. In the degree-1 case (Fig. 11a), the ROM simply stores the function values in order. Since the index of $f(x_i)$ is identical to *Seg_Addr*, $f(x_{i+1})$ is simply the next location in the ROM. Analogously

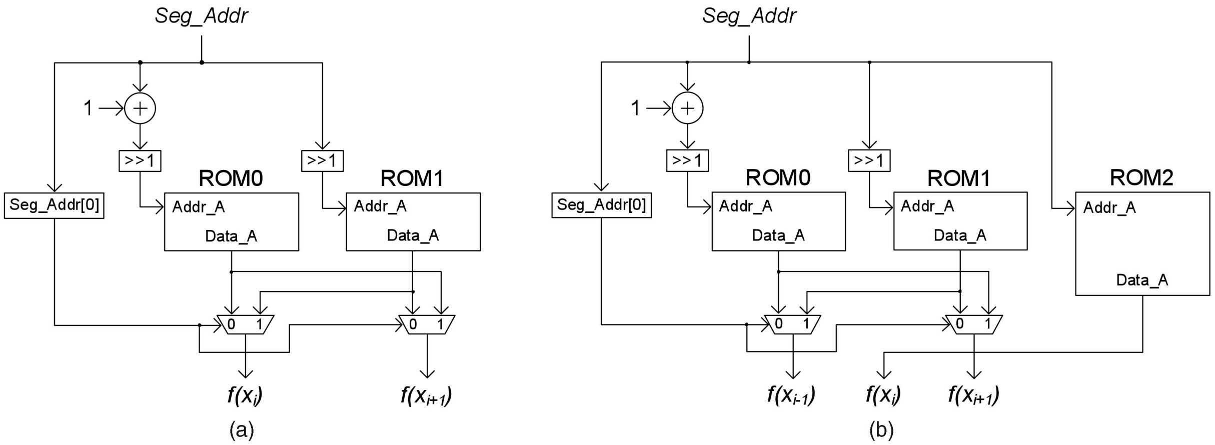


Fig. 8. Single-port ROM architectures for extracting function values for degree-1 and degree-2 method 1 interpolations. ROM0 and ROM1 are interleaved memories. (a) Degree-1. (b) Degree-2.

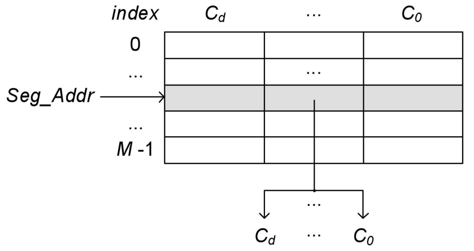


Fig. 9. Coefficient ROM organization for degree-d polynomial approximation.

to the single-port case, the dual-port degree-2 design in Fig. 11b is identical to the degree-1 design, with ROM1 storing the midpoints. Notice that, unlike the single-port architectures, the dual-port architectures do not require multiplexers at the ROM outputs.

In order to implement degree-2 method 2 interpolations, the degree-2 architectures in Figs. 8b and 11b cannot be used. Lewis [11] proposes a degree-2 method 2 architecture involving a large collection of ROMs. Each segmentation requires the function values to be partitioned into four interleaved single-port ROMs, four multiplexers, an adder, a barrel shifter, and some logic. For the $-x\ln(x)$ example in Fig. 7, since 10 outer segments are required, a total of 40 ROMs would be necessary. Although method 2 demands slightly fewer function values than method 1 (Table 1), due to the overhead of control circuitries in each ROM, the total amount of hardware required is likely more than method 1. Furthermore, technology-specific issues need to be addressed: For instance, the size of an embedded block RAM on Xilinx FPGAs is fixed at 18 Kbits. Consuming small portions of multiple memories leads to poor device utilization. Therefore, in the following discussions, we consider method 1 when targeting degree-2 interpolations.

As noted earlier, for both approximations and interpolations, the polynomial input $\tilde{x}$ is defined as $\tilde{x} = (x - x_i)/h$. This means that, for approximations and degree-1 interpolations, $\tilde{x}$ will be over the range $\tilde{x} = [0, 1)$ and, for degree-2 interpolations, $\tilde{x}$ will be over the range $\tilde{x} = [-1, 1)$. For approximations and degree-1 interpolations, this involves selecting the least significant $B_{\tilde{x}}$ bits of x that vary within the segment. For instance, in a uniform segmentation involving $2^{B_{x_{addr}}}$ segments, the most significant $B_{x_{addr}}$ bits remain constant and the remaining least significant part $\tilde{x}$ varies within a given segment. For degree-2 interpolations, the LSBs that remain constant are left shifted by 1 bit and the most significant bit (MSB) is inverted to give the range $\tilde{x} = [-1, 1)$.

The polynomial coefficients are easily computed from the function values via constant shifts and additions/subtractions by following (3) and (4) for degree-1 and (9)-(11) for degree-2. Fig. 12 depicts the coefficient computation and polynomial evaluation circuit for degree-2 interpolations. One of the major challenges of such designs is the determination of the number of bits for each operand, which will be addressed in the following section.

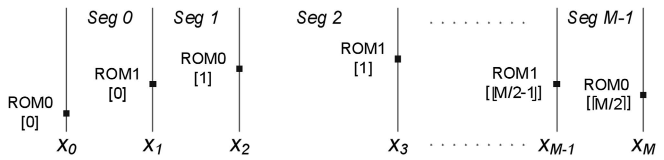


Fig. 10. Data for the interleaved memories ROM0 and ROM1 in single-port degree-1 interpolation architecture (see Fig. 8a).

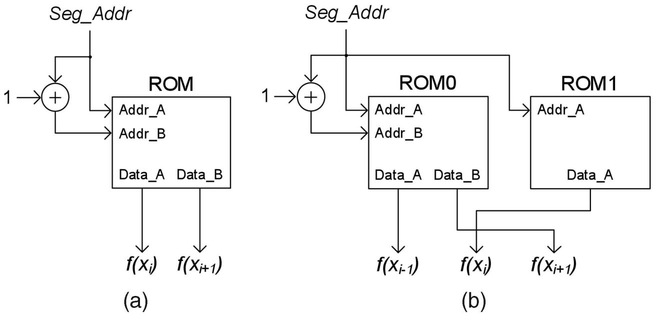


Fig. 11. Dual-port ROM architectures for extracting function values for degree-1 and degree-2 method 1 interpolations. (a) Degree 1. (b) Degree 2.

In Fig. 12, the Horner method is utilized for the evaluation of degree-2 polynomials. It has been shown in [19] that a direct evaluation of the polynomial (that is, $C_2\tilde{x}^2 + C_1\tilde{x} + C_0$) with the use of a dedicated squaring unit and a carry-save addition (CSA)-based fused-accumulation tree can lead to a more efficient VLSI implementation. However, as will be shown in Section 7, our studies show that this efficiency advantage does not necessarily hold for FPGA implementations.

6 Bit Width Optimization

For hardware implementations, it is desirable to minimize the bit widths of the coefficients and operators for area, speed, and power efficiency while respecting the error

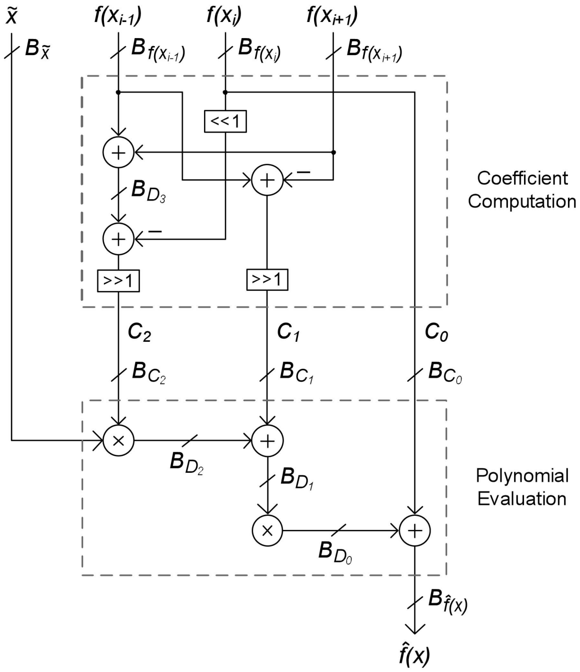


Fig. 12. Coefficient computation and polynomial evaluation circuit for degree-2 interpolations.

TABLE 2
Bit-Widths Obtained after the Bit-Width Optimization for the Degree-2 Interpolation (Fig. 12) to $\ln(1+x)$ Accurate to 16 FBs Using Hierarchical Segmentation

Operand	$\tilde{x}$	$f(x_{i-1})$	$f(x_i)$	$f(x_{i+1})$	C_2	C_1	C_0	D_3	D_2	D_1	D_0	$\hat{f}(x)$
B	15	23	23	23	9	20	23	24	20	17	23	17
(IB, FB)	(1,14)	(1,22)	(1,22)	(1,22)	(-9,18)	(-3,23)	(1,22)	(2,22)	(-9,29)	(-3,20)	(-3,26)	(1,16)

constraint at the output. Two's complement fixed-point arithmetic is assumed throughout. Given an operand z, its integer bit-width (IB) is denoted by IB_z, and its FB width is denoted by FB_z, that is, $B_z = IB_z + FB_z$. The dynamic ranges of the operands are inspected to compute the IBs, followed by a precision analysis to compute the FBs, which are based on the approach described in [29].

Many previous contributions on approximations and interpolations optimize bit widths via a dynamic method, where a group of arithmetic operators is constrained to have the same bit width [11]. The design is exhaustively simulated and the error at the output is monitored. This process is performed iteratively until the best set of bit widths is found. In contrast, the bit width methodology presented here is analytical and the bit widths are allowed to be nonuniform and are minimized via a user-defined cost function involving metrics such as circuit area. Recently, Michard et al. [30] have proposed an analytical bit width optimization approach for polynomial approximations. The work described in [30] focuses on minimizing bit widths rapidly and achieving guaranteed error bounds. In contrast, in this present work, the implementation costs of the operators and tables are considered.

6.1 Framework

To compute the dynamic range of an operand, the local minima/maxima and the minimum/maximum input values of each operand are examined. The local minima/maxima can be found by computing the roots of the derivative. Once the dynamic range has been found, the required IB can be computed trivially. Since piecewise polynomials are being targeted, the polynomial evaluation hardware needs to be shared among different sets of coefficients. The IB for each operand is found for every segment and is stored in a vector. Since the operand needs to be wide enough to avoid an overflow for data with the largest dynamic range, the largest IB in the vector is used for each operand.

There are three sources of error: 1) the inherent error ϵ_∞ due to approximating/interpolating functions with polynomials, 2) the quantization error ϵ_Q due to finite-precision effects incurred when evaluating the polynomials, and 3) the error of the final output rounding step, which can cause a maximum error of 0.5 ulp. In the worst case, ϵ_∞ and ϵ_Q will contribute additively, so, to achieve a faithful rounding, their sum must be less than 0.5 ulp. We allocate a maximum of 0.3 ulp for ϵ_∞ and the rest is for ϵ_Q, which is found to provide a good balance between the two error sources. Round to the nearest must be performed at the output $\hat{f}(x)$ to achieve a faithful rounding, but either rounding mode can be used for the operands. Since truncation results in better delay and area characteristics over round to the nearest, it is used for the operands.

Quantization errors ϵ_z for truncation and round to the nearest for an operand z are given as follows:

$$\text{Truncation}: \quad \epsilon_z = \max(0, 2^{-FB_z} - 2^{-FB_{z'}}), \tag{13}$$

$$\text{Round to the nearest}: \quad \epsilon_z = \begin{cases} 0, & \text{if } FB_z \geq FB_{z'} \\ 2^{-FB_z - 1}, & \text{otherwise,} \end{cases} \tag{14}$$

where $FB_{z'}$ is the full precision of z before quantization. For the addition $z = x + y$ and the multiplication $z = x \times y$, $FB_{z'}$ is defined as follows:

$$z = x + y: \; FB_{z'} = \max(FB_x, FB_y), \tag{15}$$

$$z = x + y: \; FB_{z'} = \max(FB_x, FB_y). \tag{16}$$

Using the equations above, an analytical error expression that is a function of the operand bit widths can be constructed at the output $\hat{f}(x)$. Simulated annealing is applied to the error expression in conjunction with a hardware area estimation function (discussed in Section 6.2).

Table 2 shows the bit-widths determined for the degree-2 interpolation (Fig. 12) of $\ln(1+x)$ accurate to 16 FBs (that is, $FB_{\hat{f}(x)} = 16$) using hierarchical segmentation. A negative IB refers to the number of leading zeros in the fractional part. For example, for the operand C_2, $IB = -9$ indicates that the first nine FBs of C_2 will always be zero. This fact can be exploited in the hardware implementation. The table indicates that the bit widths of high-degree coefficients are considerably smaller than those of low-degree coefficients, that is, $B_{C_2} < B_{C_1} < B_{C_0}$. This is always the case for both approximations and interpolations and is due to the LSBs of high-degree coefficients contributing less to the final quantized result. The table also indicates that the width of each function value is 23 bits. After hierarchical segmentation, it is found that a total of 31 function values is required. Hence, the total size of the three ROMs in Fig. 8b for this particular example is $23 \times 31 = 713$ bits.

6.2 Resource Estimation

Resource estimations are required as the cost function for the simulated annealing process performed during the bit width optimization step. More precise resource estimations will lead to the determination of a better set of bit widths. Hence, a precise resource estimation is crucial to achieving fair comparisons between optimized approximations and interpolations.

In this work, we perform a validation using Xilinx FPGAs and thus need to identify appropriate cost functions. Such cost functions are, of necessity, specifically to the target platform. For other synthesis tools and devices, other target-specific resource models can be used with appropriate modifications. The primary building block of Xilinx-Virtex-series FPGAs is the "slice," which consists of two 4-input lookup tables (except for the recently released Virtex-5, which uses 6-input lookup tables), two registers and two multiplexers, and additional circuitry such as carry logic and AND/XOR gates [31]. The four-input lookup table can also act as 16×1 RAM or a 16-bit shift register. Although recent-generation FPGAs contain substantial amounts of hardwired-dedicated RAM and dedicated multipliers, we consider implementations based on slices only in order to obtain comparisons on lookup table-based structures. The estimated FPGA area is expressed in units of slices for ROM, addition, rounding, and multiplication, which are the four major building blocks of polynomial structures. These resource estimates are only used during the design optimization process. The resource utilization results reported in Section 7 are based on experiments and measurements and not estimates.

6.2.1 Read-Only Memory

A ROM can be implemented either by logic or by configuring a series of lookup tables as memory. The size of a ROM implemented via logic is rather difficult to predict due to various logic minimizations performed during synthesis. However, it is possible to estimate the size of a lookup table ROM, which is known as the distributed ROM [32]. Since each slice can hold 32 bits in its two lookup tables, the slice count can be estimated by

$$Slices = ROM\ Size\,/\,32, \tag{17}$$

where the "ROM Size" is in bits. For instance, the degree-2 example in Table 2 with 31 function values would require $(31 \times 23)/32 \approx 23$ slices.

6.2.2 Addition

On FPGAs, additions are efficiently implemented due to the fast carry chains, which run through the lookup tables. Through the use of xor gates within the slices, two full adders can be implemented within a slice [33]. The addition $x + y$ requires $\max(B_x, B_y)$ full adders. Therefore, the slice count of an addition can be estimated by

$$Slices = \max(B_x, B_y)/2. \tag{18}$$

6.2.3 Rounding

Whereas truncation does not require any hardware, round to the nearest requires a circuitry for rounding. We perform an unbiased symmetric rounding toward $\pm\inf$, which involves adding or subtracting half an LSB of the desired output bit width. Hence, when rounding a B_x-bit number to B_y bits, where $B_x > B_y$, we simply need a B_y-bit adder. The slice count is given by

$$Slices = B_y/2. \tag{19}$$

6.2.4 Multiplication

A commonly used but sometimes inaccurate measure of the slice count of an Xilinx lookup table-based multiplication $x \times y$ is given by

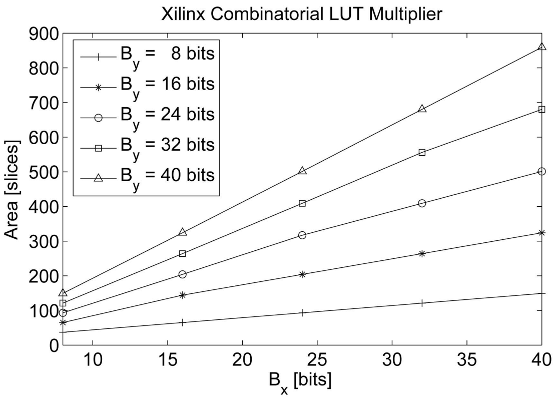


Fig. 13. Area variation of a Xilinx combinatorial LUT multiplier mapped on a Virtex-II Pro XC2VP30-7 FPGA.

$$Slices = \sum_{i=1}^{n} 2^{n-i-1}(B_x + 2^i), \tag{20}$$

where $n = \lfloor(\log_2 B_y) + 0.5\rfloor$, which defines the number of stages in the multiplier [34]. However, when B_x and B_y are not powers of two, (20) can generate misleading estimates. For instance, a 14-bit × 12-bit multiplier (that is, $n = 3$) requires 61 slices by using (20), whereas the actual mapped area on the FPGA is 93 slices, which is over 50 percent greater than the estimate.

In order to establish a more accurate measure, we have mapped various lookup table multipliers with operand sizes of 4 to 40 bits in steps of 4 bits on the FPGA and we record the slice counts. The resulting plot for steps of 8 bits is shown in Fig. 13. For any given B_x and B_y, we perform bilinear interpolation using the data points in Fig. 13. For the 14-bit × 12-bit multiplication, for instance, the interpolation results in 93 slices, which is identical to the actual mapped area. An alternative approach to estimating the multiplication area is to examine the additions involved and incorporate (18). This approach gives better estimates than (20) but is slightly inferior to the bilinear interpolation approach described above.

7 Experimental Results

The Xilinx XUP platform [35], which hosts a 0.13 μm Xilinx Virtex-II Pro XC2VP30-7 FPGA is chosen for experimental evaluation. The designs are written in behavioral VHDL. Synplicity Synplify Pro 8.6.1 is used for synthesis and Xilinx ISE 8.1.03i is used for placement and routing. The placement and routing effort level is set to "high." All designs are fully combinatorial, with no pipeline registers. The Virtex-II Pro XC2VP30-7 has a total of 13,696 slices. "Precision" refers to the number of FBs at the output $\hat{f}(x)$. With the exception of Table 3, which provides both ASIC and FPGA results, all figures and tables represent FPGA results.

Distributed ROMs are used for all of the results. However, distributed ROMs cannot realize true dual-port ROMs: They emulate dual-port functionality by replicating single-port ROMs [32], which has the obvious disadvantage of area inefficiency. Thus, the single-port ROM architectures in Fig. 8 are chosen for interpolations.

TABLE 3
ASIC and FPGA Implementation Results for 12-Bit and 20-Bit Degree-2 Approximations to $\ln(1+x)$ Using the Horner Rule and Direct Evaluation [19]

Precision	Platform	Architecture	Area	Delay
12 bits	ASIC	Horner (RCA)	3865 gates	7.4 ns
		Direct (RCA)	6207 gates	3.8 ns
		Direct (CSA)	5355 gates	3.7 ns
	FPGA	Horner (RCA)	142 slices	29.0 ns
		Direct (RCA)	206 slices	26.8 ns
		Direct (CSA)	264 slices	28.5 ns
20 bits	ASIC	Horner (RCA)	9919 gates	12.9 ns
		Direct (RCA)	17150 gates	9.0 ns
		Direct (CSA)	14729 gates	9.4 ns
	FPGA	Horner (RCA)	491 slices	40.5 ns
		Direct (RCA)	649 slices	36.1 ns
		Direct (CSA)	830 slices	39.4 ns

RCA refers to ripple-carry addition, while CSA refers to carry-save addition.

Table 3 compares the ASIC and FPGA implementation results for 12-bit and 20-bit degree-2 approximations to $\ln(1+x)$ using the Horner rule (utilizing the standard ripple-carry addition (RCA)) and the direct evaluation approach via squaring and CSA-based fused accumulation tree [19]. Results for the direct evaluation implementation using RCA for the fused accumulation tree are also provided as reference. The bit widths of the operands are optimized via the approach described in Section 6. The number of segments is identical for both the Horner and direct evaluations and the coefficient bit widths are very similar in the two approaches, thus resulting in comparable coefficient ROM sizes. The ASIC results are obtained using the UMC 0.13 μm standard cell library with Synopsys Design Compiler and Cadence Encounter. The ASIC results show that the direct approach requires more area than Horner, but approximately halves the delay. As one might expect, when targeting ASICs, CSA leads to a more efficient design than RCA. However, a somewhat different trend is observed with the FPGA results. Horner is still the best in terms of area, but the differences in delays are now very small. This is primarily due to the significance of routing delays on FPGAs. Furthermore, the CSA results are inferior to the RCA results in both area and delay. FPGAs contain dedicated fast carry chains for additions and CSA cannot exploit this feature, resulting in a less efficient implementation than the use of RCA.

Table 4 compares a uniform segmentation and a hierarchical segmentation of 16-bit degree-2 approximations to three functions over $x = [0,1)$. For $\ln(1+x)$ and $\sin(x\pi/2)$, the hierarchical segmentation results in fewer segments and less memory burden. Nevertheless, for the extra circuitry required for computing the segment address, the area on the device is larger and a minor delay penalty is introduced. For $-x\ln(x)$, however, due to its nonlinear nature, significant memory and area savings are achieved through the hierarchical segmentation, but at the expense of moderate delay overhead. Hence, for the remaining results, a uniform segmentation is used for $\ln(1+x)$ and $\sin(x\pi/2)$ and a hierarchical segmentation is used for $-x\ln(x)$.

TABLE 4
Comparisons of the Uniform Segmentation and the Hierarchical Segmentation of 16-Bit Degree-2 Approximations

Function	Segmentation	Segments	ROM Size [bits]	Area [slices]	Delay [ns]
$\ln(1+x)$	Uniform	16	768	332	34.7
	Hierarchical	13	702	411	38.9
$\sin(x\pi/2)$	Uniform	32	1,536	358	35.9
	Hierarchical	20	1,060	429	39.4
$-x\ln(x)$	Uniform	8,192	303,104	5,196	42.5
	Hierarchical	55	2,640	670	51.8

7.1 Memory

Table 5 shows the variation in the total number of segments and ROM size of approximations/interpolations. With respect to the degree-1 results, the segment counts for approximations and interpolations are identical for $\ln(1+x)$ (a uniform segmentation) and interpolations require just two segments for $-x\ln(x)$ (a hierarchical segmentation). This is due to degree-1 interpolations exhibiting comparable worst case error performance with degree-1 approximations, as discussed at the end of Section 3.2.1. The function $-x\ln(x)$ has high nonlinearities over the interval and thus requires more segments than

TABLE 5
Variation in the Total Number of Segments and ROM Size of Approximations/Interpolations

Degree	Function	Precision	Method	Segments	ROM Size [bits]	ROM Size Difference
1	$\ln(1+x)$	12	Approx	32	864	35%
			Interp	32	561	
		18	Approx	256	8,704	35%
			Interp	256	5,654	
		24	Approx	2,048	88,064	35%
			Interp	2,048	57,373	
	$-x\ln(x)$	12	Approx	94	2,162	28%
			Interp	96	1,552	
		18	Approx	766	24,512	34%
			Interp	768	16,149	
		24	Approx	6,142	251,822	34%
			Interp	6,144	165,915	
2	$\ln(1+x)$	12	Approx	8	320	4%
			Interp	16	306	
		18	Approx	32	1,664	12%
			Interp	64	1,462	
		24	Approx	128	7,808	6%
			Interp	256	7,324	
	$-x\ln(x)$	12	Approx	22	880	8%
			Interp	46	814	
		18	Approx	87	4,611	8%
			Interp	190	4,224	
		24	Approx	375	23,625	12%
			Interp	767	20,736	

TABLE 6
Memory Requirements of Different Degree-2 Architectures Using Uniform Segmentation for the Evaluation of $\sin(x)$ Accurate to 24 Fractional Bits

Architecture	Jain [36]	Piñeiro [19]	Cao [12] (Composite)	Cao [12] (Hybrid)	Proposed		
Method	Approximation	Approximation	Interpolation	Interpolation	Approximation		Interpolation
Range of x	$[0, \pi/2)$	$[0, 1)$	$[0, \pi/4)$	$[0, \pi/4)$	$[0, \pi/2)$	$[0, 1)$	$[0, \pi/4)$
Segments	256	64	128	256	128	64	128
ROM Size [bits]	14,592	3,712	3,522	11,008	8,576	4,480	3,773

$\ln(1+x)$. Our examination of other functions confirms the general trends observed in Table 5. Generally, degree-1 interpolations require in the range of 28 percent to 35 percent less ROM than approximations. This is because degree-1 interpolations need to store one function value per segment, whereas degree-1 approximations need to store two coefficients per segment. This range is generally consistent with, but at the lower end of, the range of 30 percent to 50 percent provided by Lewis, who considers only the number of memory locations [11]. In contrast, we also consider the impact of the bit widths and are thus able to narrow the range of ROM size differences.

For degree-2 designs, interpolations require about twice the number of segments as approximations. This is due to the higher errors of degree-2 interpolations compared to degree-2 minimax approximations, as discussed in Section 3.2.2. The ROM size results indicate that interpolations utilize in the range of 4 percent to 12 percent less memory than approximations (these results, like those for degree-1, generalize across other functions, in addition to the ones shown in the table). This is a much smaller difference than the reported range of 20 percent to 40 percent [11], [12] since that work was focused primarily on specific architectures, with ROM comparisons offered as estimates as part of the general discussion. In contrast, this present work specifically focuses on implementation details such as operator bit widths, which, in the degree-2 case, have the effect of dramatically reducing the ROM differences between approximation and interpolation.

Table 6 compares the memory requirements of different degree-2 architectures using a uniform segmentation for the evaluation of $\sin(x)$ accurate to 24 FBs. We are able to save significant memory over [36], mainly because, although Lagrange polynomials are used in [36], minimax polynomials are used in our work, which exhibit superior worst-case error behavior. Compared to [19], our design requires the same number of segments, but the ROM size requirement is slightly higher. In [19], an iterative bit-width optimization technique based on an exhaustive simulation is employed, which can result in smaller bit widths than our approach (Section 6) but at the expense of longer optimization times. Compared to the composite interpolation architecture by Cao et al. [12], our design leads to marginally higher memory requirements. This is because two polynomials are combined in [12], which can reduce the interpolation error slightly, but increases the computation burden on the coefficient computation (five additions/subtractions instead of three) and demands, fetching four function values instead of three. The hybrid interpolation architecture [12] stores the second-degree coefficient C_2 in the memory and the function values. This approach eliminates two additions/subtractions in the coefficient computation step but results in a threefold increase in storage requirements.

7.2 Area and Delay

Fig. 14 illustrates the hardware area variation of approximations/interpolations to $\ln(1+x)$. The upper and lower parts of each bar indicate the portion used by the computation and memory, respectively. The computation area of interpolations is slightly larger than approximations, most likely due to the overhead of the coefficient computation and the extra circuitries around the ROMs (an incrementer and two multiplexers, as shown in Fig. 8).

Fig. 15 examines the hardware area variation of various degree-1 and degree-2 approximations/interpolations. The overall trend in degree-1 designs is similar to the ROM size variation in Table 5 because the ROM size is the dominant area factor in degree-1 approximations/interpolations, as discussed earlier. The area requirements of degree-1 interpolations are similar to those of approximations at precisions below 18-20 bits. However, at higher precisions, degree-1 interpolations lead to significant area savings. Although Table 5 suggests that degree-2 interpolations require slightly less memory than approximations, the hardware area of degree-2 interpolations is, in fact, higher. This is due to the additional hardware required for transforming function values to coefficients and the extra circuitries for the ROMs.

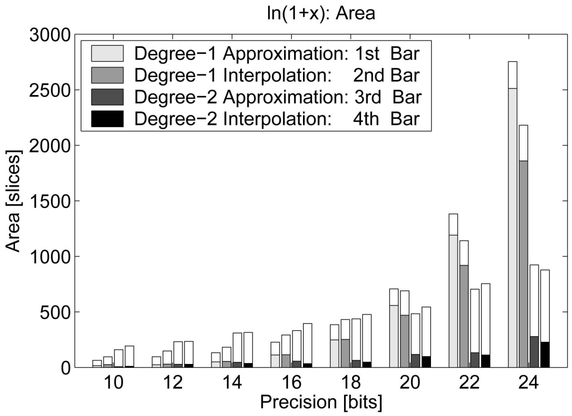


Fig. 14. Area variation of approximations/interpolations to $\ln(1+x)$ with uniform segmentation. The upper and lower parts of each bar indicate the portion used by computation and memory, respectively.

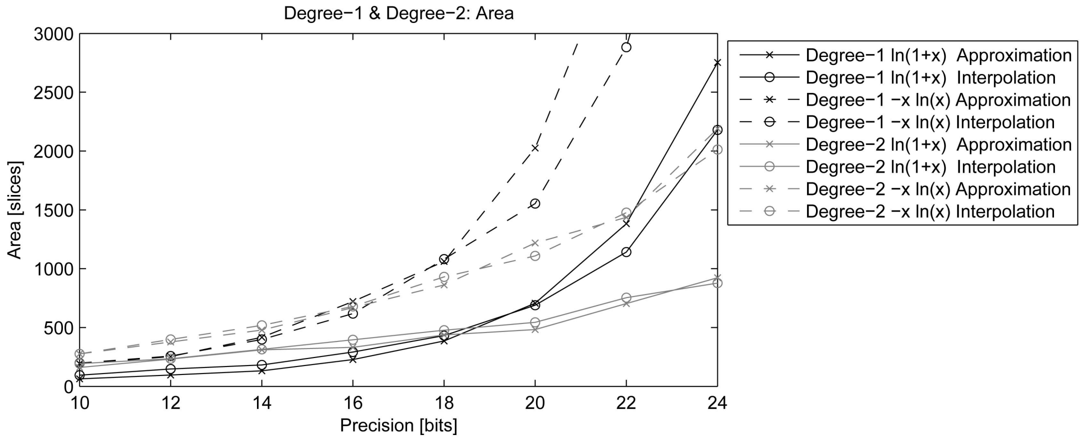


Fig. 15. Area variation of approximations/interpolations.

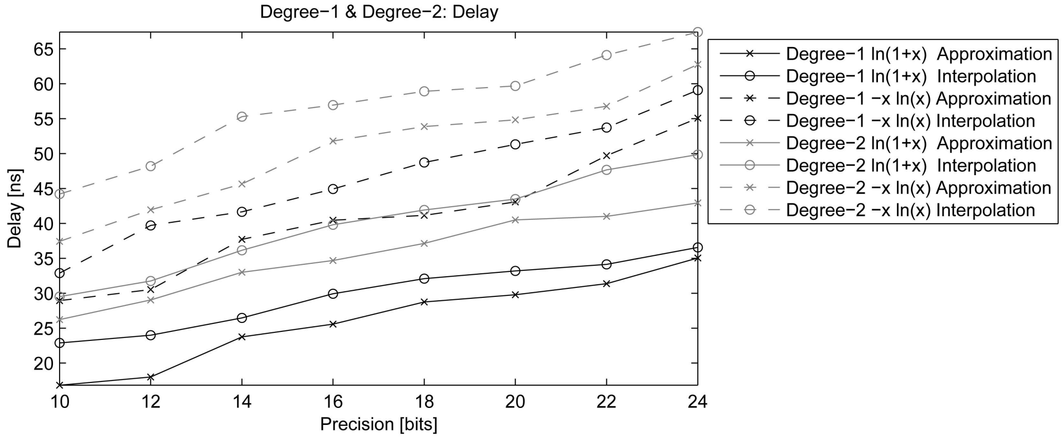


Fig. 16. Delay variation of approximations/interpolations.

Fig. 16 shows the delay variation with precision of approximations/interpolations. Interpolation generally suffers delay penalties of around 5 ns relative to approximation due to the burden of extra computation. Note that, if dedicated RAMs inside the FPGA are utilized, degree-1 designs would benefit more than degree-2 designs due to their larger memory requirements. On the other hand, if dedicated multipliers are utilized, degree-2 designs would benefit more due to their extra multiplication step.

7.3 Power

To obtain power consumption, the current flowing into the FPGA (which runs at 1.5 V) is measured via a multimeter at room temperature. A 32-bit Tausworthe uniform random number generator [37] is placed in front of each design to supply random inputs. The output bits are XORed and fed an output pin. The power results shown in this section correspond to the total power, which is the sum of two main components: static power and dynamic power. Static power largely results from the transistor leakage current, whereas dynamic power is primarily due to switching activities for charging and discharging load capacitance. Although static power is becoming increasingly significant as process geometries shrink [38], dynamic power is still the dominant factor in the 0.13 μm FPGA considered here.

Fig. 17 examines variations in the total power consumption per megahertz with precision of approximations/interpolations. The figure indicates that the total power consumption per megahertz increases exponentially with precision and the general trend tracks the area results in Fig. 15. At precisions below 14-16 bits, degree-1 designs have a modest power advantage over degree-2 designs. At higher precisions, however, degree-2 designs demand less power and the gap widens rapidly with precision. Degree-1 interpolations lead to considerable power savings over their approximation counterparts due to their smaller memory requirements.

7.4 Trade-Off Plots

Fig. 18 shows the scatter plots in the area and delay and area and power space for 12-bit approximations/interpolations, whereas Fig. 19 shows the corresponding scatter plots for a 20-bit precision. The 12-bit precision plots indicate that, depending on the function, degree-1 approximations/interpolations are the most desirable in terms of all three dimensions (area, delay, and power). The 20-bit precision plots, however, exhibit Pareto-optimal behavior: The majority of the data points are the best that can be achieved without disadvantaging at least one dimension. In both cases, the presence of results from both approximation and interpolation, as opposed to one of these methods alone, populates the space more richly and offers a wider range of design options.

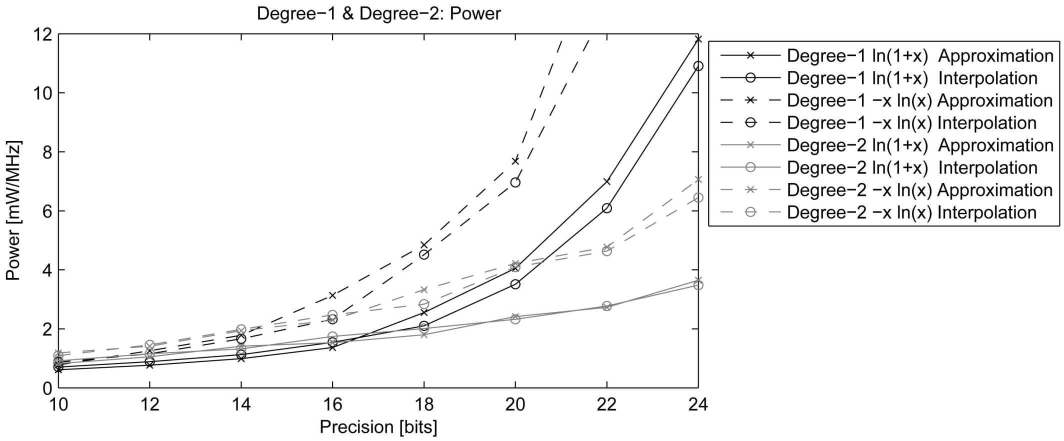


Fig. 17. Power consumption variation of approximations/interpolations.

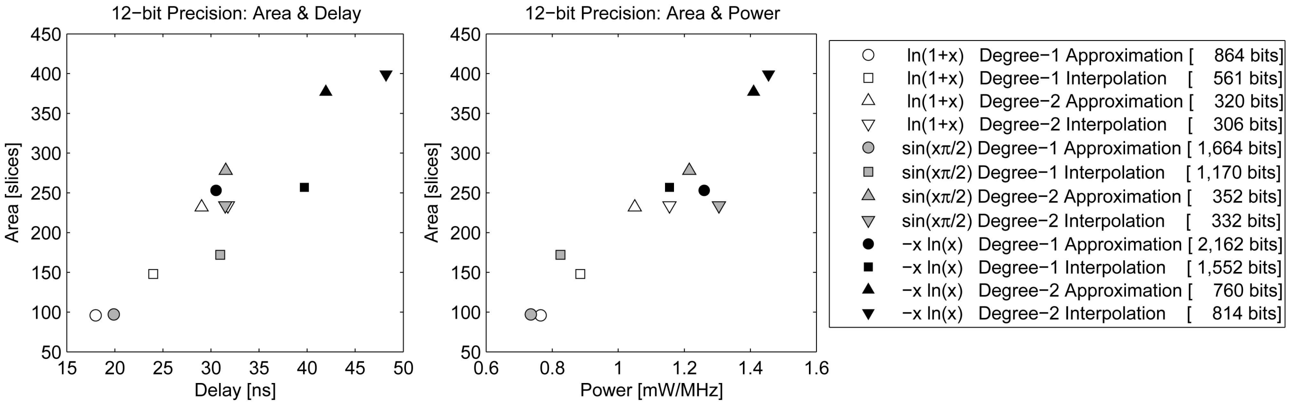


Fig. 18. Scatter plots in the area and delay and area and power space for 12-bit approximations/interpolations. Memory requirements are shown in the square brackets.

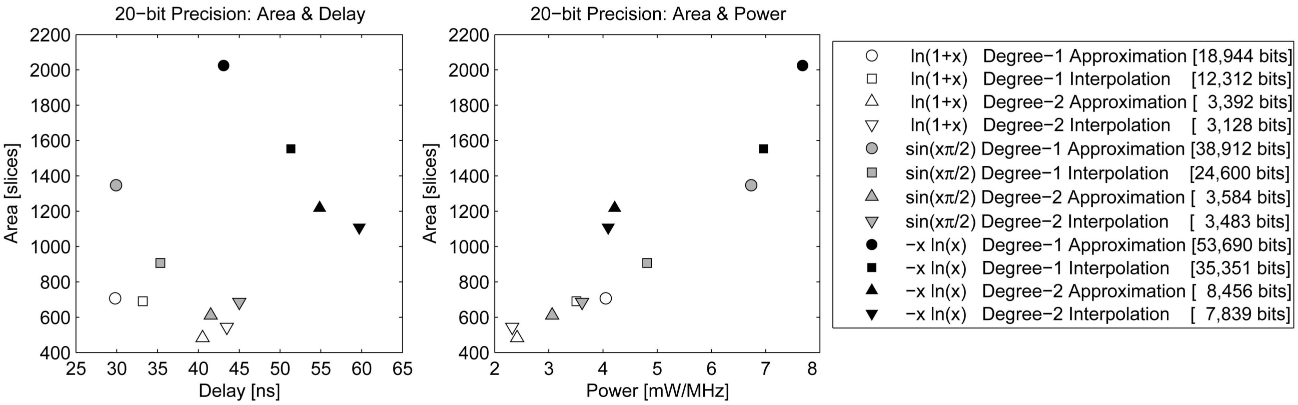


Fig. 19. Scatter plots in the area and delay and area and power space for 20-bit approximations/interpolations. Memory requirements are shown in the square brackets.

Different applications impose different constraints on area, delay, power, and memory. Hence, depending on the target function and precision requirements, the plots shown here enable the selection of the most suitable function evaluation method. If looking at single dimensions, for the 20-bit evaluation of $\ln(1+x)$, for instance, a degree-2 approximation leads to the least area, a degree-1 approximation results in the shortest combinatorial delay, whereas a degree-2 interpolation leads to the lowest memory and power requirements. Furthermore, requirements involving multiple dimensions can be satisfied at the same time: If an application imposes an area below 1,400 slices, a combinatorial delay below 55 ns, and a power consumption under 5 mW/MHz for the 20-bit evaluation of $-x\ln(x)$, then a degree-2 approximation provides a satisfactory solution.

8 Conclusions

We have examined the hardware implementation trade-offs when evaluating functions via piecewise polynomial approximations and interpolations. Hardware architectures for approximations and interpolations have been described and realized on a 0.13 μm Xilinx Virtex-II Pro FPGA. Results indicate that, although interpolations have an inherent drawback of having to compute coefficients at runtime, they have certain advantages over approximations. For degree-1 interpolations, thanks to a function value adjustment approach at the compilation time, the worst-case error behavior can be calibrated to be comparable to that of degree-1 minimax approximations. This leads to an average degree-1 memory savings of 28 percent to 35 percent over approximation. In addition, for precisions exceeding 16 bits, interpolation offers considerable savings in area and power over approximation. For degree-2, the memory size reductions of interpolation over approximation are in the range of only 4 percent to 12 percent, which are significantly lower than what is commonly assumed. The degree-2 hardware area and power consumption for approximation and interpolation is found to be similar. The methodology presented here can be used to produce area, delay, and power trade-off plots, thereby enabling the choice of the most suitable function evaluation method for a given application requirement.

Acknowledgments

The authors thank Hyungjin Kim and David Choi for their assistance. The support of the US Office of Naval Research under Contract N00014-06-1-0253, the US National Science Foundation under Grants CCR-0120778 and CCF-0541453, the Croucher Foundation, Xilinx Inc., and the UK Engineering and Physical Sciences Research Council under Grants EP/C509625/1, EP/C549481/1, and GR/R 31409 is gratefully acknowledged. Dong-U Lee was with the Electrical Engineering Department at the University of California, Los Angeles when this work was conducted.

References

[1] Y. Song and B. Kim, "Quadrature Direct Digital Frequency Synthesizers Using Interpolation-Based Angle Rotation," *IEEE Trans. VLSI Systems,* vol. 12, no. 7, pp. 701-710, 2004.

[2] H. Shin, J. Lee, and J. Kim, "A Hardware Cost Minimized Fast Phong Shader," *IEEE Trans. VLSI Systems,* vol. 9, no. 2, pp. 297-304, 2001.

[3] K. Karagianni, V. Paliouras, G. Diamantakos, and T. Stouraitis, "Operation-Saving VLSI Architectures for 3D Geometrical Transformations," *IEEE Trans. Computers,* vol. 50, no. 6, pp. 609-622, June 2001.

[4] P. Liu and S. Bhatt, "Experiences with Parallel N-Body Simulation," *IEEE Trans. Parallel and Distributed Systems,* vol. 11, no. 12, pp. 1306-1323, Dec. 2000.

[5] Y. Hu, "CORDIC-Based VLSI Architectures for Digital Signal Processing," *IEEE Signal Processing Magazine,* vol. 9, no. 3, pp. 17-34, 1992.

[6] K. Johansson, O. Gustafsson, and L. Wanhammar, "Approximation of Elementary Functions Using a Weighted Sum of Bit-Products," *Proc. IEEE Int'l Symp. Circuits and Systems,* pp. 795-798, 2006.

[7] J.-M. Muller, *Elementary Functions: Algorithms and Implementation,* second ed. Birkhauser, 2006.

[8] F. de Dinechin and A. Tisserand, "Multipartite Table Methods," *IEEE Trans. Computers,* vol. 54, no. 5, pp. 319-330, May 2005.

[9] *Numerical Analysis,* Encyclopedia Britannica Online, http://www.search.eb.com/eb/article-235500, 2006.

[10] M. Ercegovac and T. Lang, *Digital Arithmetic.* Morgan Kaufmann, 2004.

[11] D. Lewis, "Interleaved Memory Function Interpolators with Application to an Accurate LNS Arithmetic Unit," *IEEE Trans. Computers,* vol. 43, no. 8, pp. 974-982, Aug. 1994.

[12] J. Cao, B. Wei, and J. Cheng, "High-Performance Architectures for Elementary Function Generation," *Proc. 15th IEEE Symp. Computer Arithmetic,* pp. 136-144, 2001.

[13] D. Lee, A. Abdul Gaffar, O. Mencer, and W. Luk, "Optimizing Hardware Function Evaluation," *IEEE Trans. Computers,* vol. 54, no. 12, pp. 1520-1531, Dec. 2005.

[14] R. Michard, A. Tisserand, and N. Veyrat-Charvillon, "Small FPGA Polynomial Approximations with 3-Bit Coefficients and Low-Precision Estimations of the Powers of X," *Proc. 16th IEEE Int'l Conf. Application-Specific Systems, Architecture and Processors,* pp. 334-339, 2005.

[15] A. Noetzel, "An Interpolating Memory Unit for Function Evaluation: Analysis and Design," *IEEE Trans. Computers,* vol. 38, no. 3, pp. 377-384, Mar. 1989.

[16] N. Takagi, "Powering by a Table Look-Up and a Multiplication with Operand Modification," *IEEE Trans. Computers,* vol. 47, no. 11, pp. 1216-1222, Nov. 1998.

[17] M. Arnold and M. Winkel, "A Single-Multiplier Quadratic Interpolator for LNS Arithmetic," *Proc. 19th IEEE Int'l Conf. Computer Design,* pp. 178-183, 2001.

[18] J. Detrey and F. de Dinechin, "Table-Based Polynomials for Fast Hardware Function Evaluation," *Proc. 16th IEEE Int'l Conf. Application-Specific Systems, Architecture and Processors,* pp. 328-333, 2005.

[19] J. Piñeiro, S. Oberman, J. Muller, and J. Bruguera, "High-Speed Function Approximation Using a Minimax Quadratic Interpolator," *IEEE Trans. Computers,* vol. 54, no. 3, pp. 304-318, Mar. 2005.

[20] M. Schulte and E. Swartzlander Jr., "Hardware Designs for Exactly Rounded Elementary Functions," *IEEE Trans. Computers,* vol. 43, no. 8, pp. 964-973, Aug. 1994.

[21] E.G. Walters III and M. Schulte, "Efficient Function Approximation Using Truncated Multipliers and Squarers," *Proc. 17th IEEE Symp. Computer Arithmetic,* pp. 232-239, 2005.

[22] H. Aus and G. Korn, "Table-Lookup/Interpolation Function Generation for Fixed-Point Digital Computations," *IEEE Trans. Computers,* vol. 18, pp. 745-749, 1969.

[23] V. Paliouras, K. Karagianni, and T. Stouraitis, "A Floating-Point Processor for Fast and Accurate Sine/Cosine Evaluation," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing,* vol. 47, no. 5, pp. 441-451, 2000.

[24] J. McCollum, J. Lancaster, D. Bouldin, and G. Peterson, "Hardware Acceleration of Pseudo-Random Number Generation for Simulation Applications," *Proc. 35th IEEE Southeastern Symp. System Theory,* pp. 299-303, 2003.

[25] P. Lamarche and Y. Savaria, "VHDL Source Code Generator and Analysis Tool to Design Linear Interpolators," *Proc. First IEEE Northeast Workshop Circuits and Systems,* pp. 69-72, 2003.

[26] J. Rice, *The Approximation of Functions,* vol. 1. Addison-Wesley, 1964.

[27] J. Mathews, *Numerical Methods for Mathematics, Science, and Engineering.* Prentice Hall, 1992.

[28] D. Lee, W. Luk, J. Villasenor, and P. Cheung, "Hierarchical Segmentation Schemes for Function Evaluation," *Proc. IEEE Int'l Conf. Field-Programmable Technology,* pp. 92-99, 2003.

[29] D. Lee and J. Villasenor, "A Bit-Width Optimization Methodology for Polynomial-Based Function Evaluation," *IEEE Trans. Computers,* vol. 56, no. 4, pp. 567-571, Apr. 2007.

[30] R. Michard, A. Tisserand, and N. Veyrat-Charvillon, "Optimisation d'Opérateurs Arithmétiques Matériels à Base d'Approximations Polynomiales," *Proc. Symp. Architectures Nouvelles de Machine,* pp. 130-141, 2006.

[31] C. Maxfield, *The Design Warrior's Guide to FPGAs.* Newnes, 2004.

[32] *Using Look-Up Tables as Distributed RAM in Spartan-3 Generation FPGAs (Xilinx Application Note: XAPP464).* Xilinx, http://www.xilinx.com/bvdocs/appnotes/xapp464.pdf, 2005.

[33] *Design Tips for HDL Implementation of Arithmetic Functions: Xilinx Application Note: XAPP215,* Xilinx, http://www.xilinx.com/bvdocs/appnotes/xapp215.pdf, 2000.

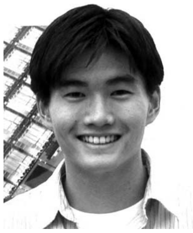
[34] *Variable Parallel Virtex Multiplier V2.0: Xilinx Logicore Product Specification.* Xilinx, http://www.xilinx.com, 2000.

[35] *Xilinx Univ. Program Virtex-II Pro Development System: Hardware Reference Manual,* Xilinx, http://www.xilinx.com/univ/xupv2p.html, 2005.

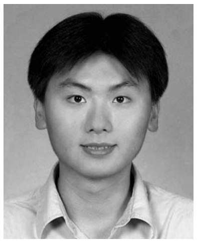
[36] V. Jain, S. Wadekar, and L. Lin, "A Universal Nonlinear Component and Its Application to WSI," *IEEE Trans. Components, Hybrids, and Manufacturing Technology,* vol. 16, no. 7, pp. 656-664, 1993.

[37] P. L'Ecuyer, "Maximally Equidistributed Combined Tausworthe Generators," *Math. Computation,* vol. 65, no. 213, pp. 203-213, 1996.

[38] "Power Consumption in 65 nm FPGAs," Xilinx White Paper WP246, Xilinx, http://www.xilinx.com/bvdocs/whitepapers/wp246.pdf, 2006.



Dong-U Lee received the BEng degree in information systems engineering and the PhD degree in computing from the Imperial College London in 2001 and 2004, respectively. From 2005 to 2007, he was a postdoctoral researcher in the Department of Electrical Engineering at the University of California, Los Angeles (UCLA), where he developed high-performance hardware designs for wireless communications and mathematical function evaluations. He is now a research scientist at Mojix Inc., Los Angeles, specializing in the hardware implementation aspects of RFID receivers. His research interests include computer arithmetic, communications, design automation, reconfigurable computing, and video image processing. He is a member of the IEEE.



Ray C.C. Cheung received the BEng degree in computer engineering and the MPhil degree in computer science and engineering from the Chinese University of Hong Kong (CUHK) in 1999 and 2001, respectively, and the PhD degree in computing from the Imperial College London in 2007. From January 2002 to December 2003, he was an instructor in the Department of Computer Science and Engineering at CUHK. He was with Stanford University and the University of California, Los Angeles (UCLA) in 2005 and 2006 as a visiting scholar. He worked as a postdoctoral researcher in the Department of Electrical Engineering at UCLA. He is currenlty a research engineer at Solomon Systech Limited, Hong Kong. His current research interests are reconfigurable computing and computer arithmetic hardware designs. He is a member of the IEEE.

Wayne Luk received the MA, MSc, and DPhil degrees in engineering and computer science from the University of Oxford, United Kingdom. He is currently a professor of computer engineering in the Department of Computing at Imperial College London and a visiting professor iat Stanford University, California, and Queen's University, Belfast, United Kingdom. His research interests include the theory and practice of customizing hardware and software for specific application domains such as multimedia, communications, and finance. Much of his current work involves high-level compilation techniques and tools for parallel computers and embedded systems, particularly those containing reconfigurable devices such as field-programmable gate arrays. He is a senior member of the IEEE.

John D. Villasenor received the BS degree in electrical engineering from the University of Virginia in 1985 and the MS and PhD degrees in electrical engineering from Stanford University in 1986 and 1989, respectively. From 1990 to 1992, he was with the Radar Science and Engineering Section at the Jet Propulsion Laboratory, Pasadena, California, where he developed methods for imaging the Earth from space. In 1992, he joined the Department of Electrical Engineering at the University of California, Los Angeles (UCLA), where is currently a professor and was the vice chair of the department from 1996 to 2002. His research interests include communications, computing, imaging and video compression, and networking. He is a senior member of the IEEE.