Optimizing Hardware Function Evaluation

Dong-U Lee, *Member, IEEE,* Altaf Abdul Gaffar, *Member, IEEE,*
Oskar Mencer, *Member, IEEE,* and Wayne Luk, *Member, IEEE*

***Abstract*— We present a methodology and an automated system for function evaluation unit generation. Our system selects the best function evaluation hardware for a given function, accuracy requirements, technology mapping and optimization metrics, such as area, throughput and latency. Function evaluation $f(x)$ typically consists of range reduction, and the actual evaluation on a small convenient interval such as $[0, \pi/2)$ for $\sin(x)$. We investigate the impact of hardware function evaluation with range reduction for a given range and precision of x and $f(x)$ on area and speed. An automated bit-width optimization technique for minimizing the sizes of the operators in the data paths is also proposed. We explore a vast design space for fixed-point $\sin(x)$, $\log(x)$ and $\sqrt{x}$ accurate to one unit in the last place using MATLAB and ASC, A Stream Compiler for Field-Programmable Gate Arrays (FPGAs). In this study, we implement over 2000 placed-and-routed FPGA designs, resulting in over 100 million Application-Specific Integrated Circuit (ASIC) equivalent gates. We provide optimal function evaluation results for range and precision combinations between 8 and 48 bits.**

***Index Terms*— Computer arithmetic, elementary function approximation, gate arrays, minimax approximation and algorithms, optimization.**

I. Introduction

FUNCTION evaluation can often be the performance bottleneck of many important compute-bound applications. Examples include elementary functions such as $\log(x)$ and compound functions such as $\sqrt{-\log(x)}$. Computing these functions quickly and accurately is a major goal in computer arithmetic and hardware design in general. Software implementations are often too slow for numerically intensive or real-time applications. For instance, over 60% of the total run time is spent on function evaluation operations in a simulation of a jet engine reported by O'Grady and Wang [1]. The performance of such applications depends on the design of an efficient hardware function evaluator. Yet in order to implement function evaluation efficiently, the hardware designer is faced with a multitude of function evaluation methods such as polynomial approximation, or table lookup combined with polynomial approximation [2]. The challenge is to provide a programming tool or library, that delivers the optimal hardware function evaluation unit for a given function, with the associated input/output range and precision, and optimization metric.

Manuscript received ————

D. Lee is with the Electrical Engineering Department, University of California, Los Angeles, USA (e-mail: dongu@icsl.ucla.edu).

A. Abdul Gaffar is with the Department of Electrical and Electronic Engineering, Imperial College, London, United Kingdom (e-mail: altaf.gaffar@imperial.ac.uk).

O. Mencer and W. Luk are with the Department of Computing, Imperial College, London, United Kingdom (e-mail: {o.mencer, w.luk}@imperial.ac.uk).

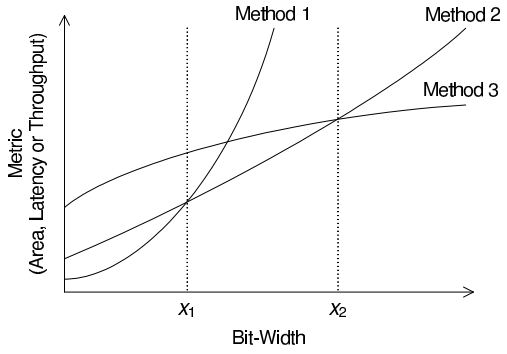

Fig. 1. Some approximation methods are better than others for a given metric at different bit-widths.

For a given accuracy requirement it is possible to plot the area, latency, and throughput tradeoff and thus identify the optimal function evaluation method. The optimality depends on further requirements such as available area, required latency and throughput. For instance, consider Fig. 1. In order to minimize the metric (e.g. area or latency), one should use method 1 for bit-widths lower than x_1, method 2 for bit-widths between x_1 and x_2, and method 3 for bit-widths greater than x_2.

Our approach explores, for a given function, seven different dimensions in optimizing hardware function evaluation: range, precision, method, hardware optimization, area, latency and throughput. The main achievements of this paper are:

- Methodology for automated function evaluation unit generation to select optimal function evaluation hardware based on a parameterized library.
- Framework for hardware function evaluation with range reduction for $\sin(x)$, $\log(x)$ and $\sqrt{x}$.
- Algorithmic design space exploration using MATLAB to guide the hardware design process in ASC.
- Bit-width optimization of the operators in the data paths using a binary search technique in MATLAB.
- Vast hardware design space exploration of over 2000 FPGA designs on area, latency and throughput using ASC.

The rest of this paper is organized as follows. Section II covers background material and related work. Section III provides an overview of our approach. Section IV describes range reduction and its application to the three functions presented in this paper. Section V examines the degrees of freedom in hardware function evaluation. Section VI describes how we explore the algorithmic side of the design space and automate the generation of hardware designs. Section VII explains how

the bit-widths of the operators in the data paths are optimized. Section VIII presents our framework for hardware design space exploration. Section IX discusses results, and Section X offers conclusions and thoughts on future work.

II. Background

There are numerous methods to approximate a function over a given interval, and the optimal method depends on the precisions of the inputs and outputs as studied in [3]. Yet, we are not aware of any other work that attempts to guide the designer as to which method is optimal for a particular case. Direct table lookups are impractical for precisions higher than a few bits, since table size increases exponentially with the input size. Symmetric table addition methods [4] are fast with moderate table sizes for precisions lower than 20 bits, but are perhaps inappropriate for larger precisions due to its large table sizes. Function evaluations using CORDIC [5] provides a popular research topic, involving only shift and add operations. However, CORDICs have an execution time which is linearly proportional to the number of bits in the operands, and is not suitable for applications high accuracy and speed. Of course, the tradeoffs depend on the optimization metric as well.

Function evaluation typically consists of range reduction and the actual function approximation over a small interval. Range reduction [2] is crucial, since function approximation is rather limited without it, and numerous applications have a large dynamic range. However, there has been a lack of attention on hardware implementation of function approximation with range reduction for different ranges, precisions and approximation methods. To the best of our knowledge, this is the first work that deals with this important issue. We show that input and output ranges form another consideration when choosing the optimal method. Our approach is demonstrated with polynomial-only and table+polynomial methods with a varying number of polynomial coefficients.

Peymandoust and De Micheli [6] use symbolic computer algebra to optimize arithmetic data paths. Symbolic manipulations such as tree-height-reduction, factorization, expansions, and Horner transformation are incorporated to produce minimal area or minimal delay data flow designs. The main difference between their work and ours is that we consider function evaluation units with range reduction rather than just arithmetic data paths. In addition, we explore the tradeoffs of using memory and polynomials instead of just polynomials. However, their work is in some sense orthogonal to ours in that an optimal system would combine the results of the two works.

We choose hardware designs based on FPGAs to demonstrate our approach, due to their flexibility and speed. The fundamental building block of Xilinx FPGAs is the logic cell [7]. A logic cell comprises a 4-input lookup table, which can also act as a 16×1 RAM or a 16-bit shift register, a multiplexer and a register. A simplified view of a logic cell is depicted in Fig. 2. Two logic cells are paired together in an element called a slice. A slice contains additional resources such as multiplexors and carry logic to increase the efficiency of the architecture. These extra resources are equivalent to having more logic cells, and therefore a slice is counted as being equivalent of 2.25 logic cells. Recent-generation reconfigurable hardware has a large amount of slices. For instance, the Xilinx Virtex-4 XC4VLX200-11 FPGA [8], which we use to obtain our results, has 89088 slices (200448 logic cells), equivalent to over six million ASIC gates.

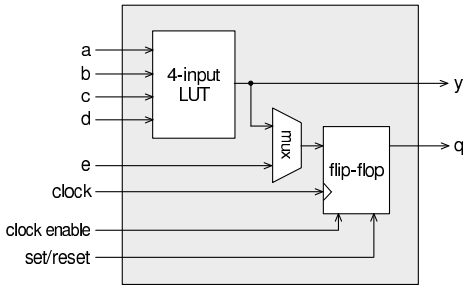

Fig. 2. Simplified view of a Xilinx logic cell. A single slice is equivalent to 2.25 logic cells.

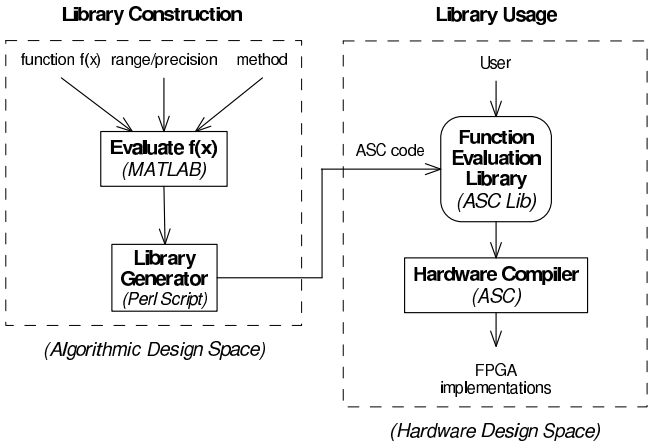

Fig. 3. Design flow: MATLAB generates all the ASC code for the library. The user simply indexes into the library with range and precision values to obtain the specific function evaluation unit.

III. Overview

Fig. 3 shows the design flow of our automated hardware function evaluation approach. The function of interest, its range and precision, and evaluation method are supplied to our MATLAB program, which automatically designs the function approximator and produces its hardware description. In our case, MATLAB produces code for ASC, A Stream Compiler for FPGAs [9]. This large collection of ASC functions is then transformed by a Perl script into an ASC function evaluation library (ASC lib). ASC then takes care of design space exploration on the architecture level, the arithmetic level, and the gate level of abstraction. The result is an optimized function evaluation library for computing with FPGAs. Device independent results at the algorithmic level can be obtained with MATLAB, and device specific results on FPGAs can be obtained with ASC as they will be discussed in Section IX.

Sign-magnitude fixed-point representation is used throughput this paper, since it allows easier manipulation of numbers

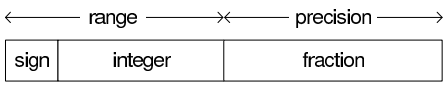

Fig. 4. The sign-magnitude fixed-point representation used in this work.

compared to two's complement. We define the sign bit and the integer bits to be the range, and the fractional bits to be the precision (Fig. 4). Ranges of 4, 8, 12, 16, 20 and 24 bits, and the same set of bits for precisions are explored. These range/precision sets result in 36 different fixed-point formats.

Given a function $f(x)$ and an interval $[a, b]$ we approximate the function with polynomials and tables. Tasks in designing a function evaluation library include automating the selection of range reduction, the selection and design of the function evaluation method, and area, latency and throughput optimizations on the lower levels of abstraction. The central contribution of this paper lies in reconsidering the above structure for user-defined fixed-point bit-widths. When implementing hardware designs, one can select any bit-width for the range and the precision of the fixed-point number. As a consequence, a function evaluation library obtains the range and precision of the input and can use this information to produce an optimized function evaluation unit. Previous work [3] shows the subproblem of how to select function evaluation methods based on precision. Based on input range and precision, we now have the following degrees of freedom:

1) applicability of range reduction
2) evaluation method selection
3) evaluation method design
   - find minimal bit-widths
   - find minimal polynomial degree (for polynomial-only method)
   - find minimal segments (for table+polynomial method)
4) optimize: area, latency or throughput

The polynomial-only (*po*) approach approximates the interval with a single polynomial, whereas the table+polynomial (*tp*) approach performs piecewise polynomial approximation with equally sized segments. The ASC function evaluation library takes the range, precision and optimization metric, and instantiates one of many instances of the corresponding function evaluation unit.

In this paper, the outputs of our function evaluation units are accurate to one unit in the last place (ulp). Assume we require a hardware unit to compute $\sin(x)$, where x is a fixed-point number with four range bits and eight precision bits. Then the range of the input is $(-8, 8)$ and the expected range of the output is $[-1, 1]$. The same precision is used at the output as at the input. Hence for this example, since the precision is eight bits, the maximum absolute error of the output needs to be 2^{-8} or less to guarantee faithful rounding. The term faithful rounding is first introduced in [10], meaning that the results are rounded to the nearest or next nearest, thus accurate to one ulp.

TABLE I
RANGE REDUCTION PROPERTIES OF THE THREE FUNCTIONS.

Function	Range Reduction Type	Range Reduced Interval	Interval Size	Maximum Derivative
$\sin(x)$	Additive	$[0, \pi/2)$	$\pi/2$	1
$\log(x)$	Multiplicative	$[0.5, 1)$	0.5	1.8
$\sqrt{x}$	Multiplicative	$[0.25, 1)$	0.75	4

IV. RANGE REDUCTION

Consider an elementary function $f(x)$, where x and $f(x)$ have a given range $[a, b]$ and precision requirement. The evaluation $f(x)$ typically consists of three steps [2]:

(1) range reduction, reducing x over the interval $[a, b]$ to a more convenient y over a smaller interval $[a', b']$,

(2) function approximation on the reduced interval, and

(3) range reconstruction: expansion of the result back to the original result range.

There are two main types of range reduction:

- additive reduction: y is equal to $x - mC$;
- multiplicative reduction: y is equal to x/C^m

where integer m and a constant C are defined by the evaluated function.

We use ASC code notation in Fig. 5 to show various methods of function evaluation including range reduction and range reconstruction, which follow the ideas presented in [11] and [12]. The notations $x.range$ and $x.prec$ refer to the number of bits used for the range and precision of x, respectively. The code in Fig. 5 shows us an example a different function evaluation methods for each function. In reality, we create many combinations of evaluation methods and functions.

Table I summarizes the range reduction properties of the three functions. Equally sized segments for the table+polynomial method are employed, meaning that the approximation interval needs to be a power of two. Hence, for $\sin(x)$ we approximate over $[0, 2)$, for $\sqrt{x}$, we split the interval into two sub-intervals: $[0.25, 0.5)$ and $[0.5, 1)$. The table shows range reduced approximation interval sizes of the three functions. The larger the approximation interval, the more hardware resources are potentially required. The first order absolute maximum derivatives give us an indication of the non-linearities: more resources are required to approximate non-linear functions with large derivatives.

Fig. 6 highlights the functions over the range reduced intervals. We observe that the functions have a relatively linear behavior over these intervals, making them feasible to approximate using *po* or *tp* with equally sized segments.

V. DEGREES OF FREEDOM

This section describes the degrees of freedom a designer is faced with, when implementing function evaluation in hardware. The applicability of range reduction, approximation method selection and its design, and hardware optimizations are discussed.

Evaluating $f(x) = \sin(x)$

```
// Range Reduction
x1 = abs(x) % (2*pi);
x2 = IF(x1>pi, x1-pi, x1);
y = IF(x2>(pi/2), pi-x2, x2);

// Approximation
// g(y) where y = [0,pi/2)
// e.g. polynomial-only (po)
g = (a*y+b)*y+c;

// Range Reconstruction
f = IF(x1>pi, g, -g);
```

Evaluating $f(x) = \log(x)$

```
// Range Reduction
exp = leading_one_detect(x)-x.prec;
y = x >> exp;

// Approximation
// g(y) where y = [0.5,1)
// e.g. table+degree-1-polynomial (tp1)
g = table1[y]*y+table2[y];

// Range Reconstruction
f = g+exp*log(2);
```

Evaluating $f(x) = \sqrt{x}$

```
// Range Reduction
exp = leading_one_detect(x)-x.prec;
x1 = x >> exp;
y = IF(exp[0], x1 >> 1, x1);

// Approximation
// g(y) where y = [0.25,1)
// e.g. table+degree-2-polynomial (tp2)
g = (table1[y]*y+table2[y])*y+table3[y];

// Range Reconstruction
exp1 = IF(exp[0], exp+1 >> 1, exp >> 1);
f = g << exp1;
```

Fig. 5. Description of range reduction, approximation method and range reconstruction for the three functions $\sin(x)$, $\log(x)$ and $\sqrt{x}$.

A. Applicability of Range Reduction

Given a particular function that we want to evaluate, we can decide whether it is necessary to implement range reduction or not. In order to make the correct decision we need to consider the optimization metric (area, latency or throughput), design a function evaluation unit with and without range reduction, and select the optimal one. A preliminary study of the applicability of range reduction has been conducted in [13].

B. Approximation Method Selection

There are many possible function evaluation methods, such as symmetric table addition methods, CORDIC, rational approximation [14], polynomial-only methods and table+polynomial methods. In this paper, we explore six methods: polynomial-only (*po*) and table+polynomial methods with polynomials of degree two to six (*tp2-6*). The polynomials are

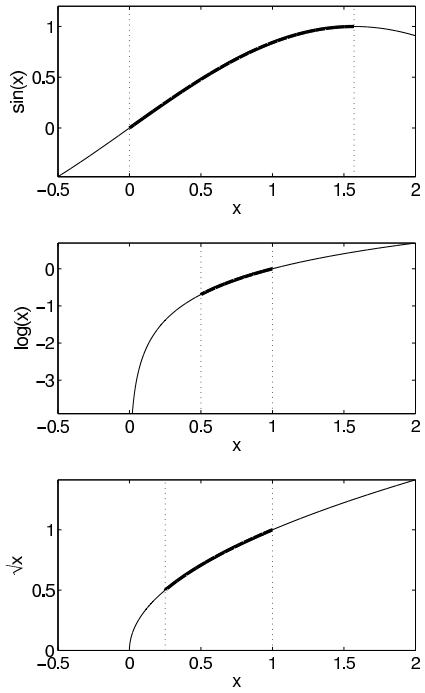

Fig. 6. Plots of the three functions over $x = [-0.5, 2]$. Range reduced intervals for each function are shown in thick lines.

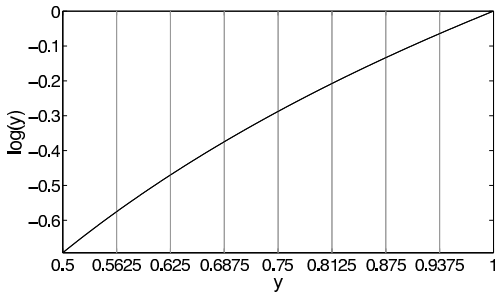

Fig. 7. Segmentation for evaluating $\log(y)$ with eight uniform segments. The leftmost (MSB) three bits of the inputs are used as the segment index.

of the form

$$g(y) = c_d y^d + c_{d-1} y^{d-1} + \ldots + c_1 y + c_0. \quad (1)$$

We use Horner's rule [2] to reduce the number of multiplications:

$$g(y) = ((c_d y + c_{d-1})y + \ldots)y + c_0 \quad (2)$$

where y is the input, d is the polynomial degree and c are the coefficients. For the table+polynomial (*tp*) approach, the input interval is split into 2^k equally sized segments. The k leftmost bits of the argument y serve as the index into the table, which holds the coefficients for that particular interval. For the polynomial-only approach, there is just one entry in the table holding the coefficients, hence no index bits are needed. Segmentation for evaluating $\log(y)$ with eight uniform segments ($k = 3$) is illustrated in Figure 7. We observe that the range reduced interval is relatively linear, and hence the use of uniform segmentation is sufficient.

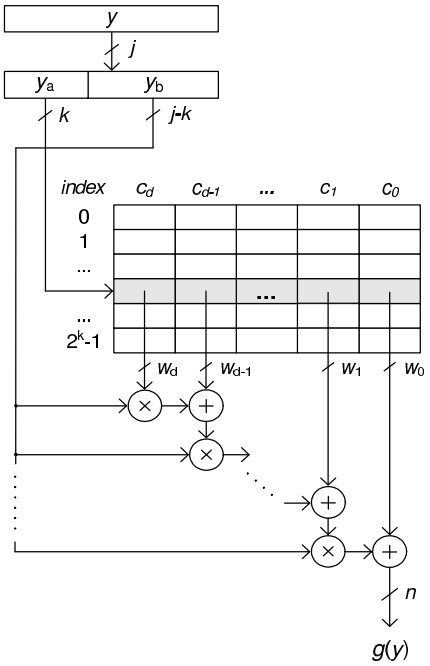

Fig. 8. Architecture of our table+polynomial (*tp*) approximation unit for degree d polynomials. w_i is the bit-width of the polynomial coefficient c_i, where $i = 0, \ldots, d$.

The architecture for an approximation unit with a *tp* scheme is depicted in Fig. 8. The *tp* methods trade off table area versus polynomial area. A multiply-and-add based tree structure can be observed, which follows Horner's rule. The polynomial coefficients are found in a minimax sense that minimize the maximum absolute error [2]. With this architecture, we need $d + 1$ table lookups, d multiplications and d additions. The size of the lookup table is given by

$$\text{table size} = 2^k \times \sum_{i=0}^{d} w_i \text{ bits.} \tag{3}$$

C. Evaluation Method Design

Once we know which method to use, we need to design the optimized unit. For the polynomial-only (*po*) method, we find the minimal degree of the polynomial that will satisfy the required output precision. For the table+polynomial (*tp*) methods, we find the minimal number of segments 2^k required that satisfy output precision requirement. We further need to determine the optimized bit-widths of the computation inside the function evaluation units for all the methods. The heuristics used for *po* and *tp* methods have linear complexities whereas the bit-width optimization process has logarithmic complexity with respect to the desired precision. These are discussed in Section VI and Section VII.

D. Optimize: Area, Latency or Throughput

While the options or selections of the previous degrees of freedom are pre-computed with MATLAB, the area, latency and throughput optimizations on the arithmetic and gate-levels can be left for the hardware compiler (ASC) to deal with. Section VIII describes how this is achieved.

```
// for a given function f, polynomial degree d
// method m, input format i=[x.range,x.prec]

if (m=='po') // for polynomial-only (po)
        // find minimal polynomial degree
        min_degree = find_min_degree(f,i);
        // optimize bit-widths
        bw = optimize_bw(f,i,min_degree);
        // generate polynomial coefficients
        coeffs = gen_coeffs(f,i,min_degree,bw);
        // generate ASC code
        gen_ASC(f,i,min_degree,bw,coeffs);

elseif (m=='tp') // for table+polynomial (tp)
        // find minimal number of segments
        min_segs = find_min_segs(f,i,d);
        // optimize bit-widths
        bw = optimize_bw(f,i,d,min_segs);
        // generate coefficient lookup table
        table = gen_table(f,i,d,min_segs,bw);
        // generate ASC code
        gen_ASC(f,i,d,min_segs,bw,table);
end
```

Fig. 9. Structure of our MATLAB tool for algorithmic design space exploration and ASC code generation.

VI. Algorithmic Design Space Exploration

We use MATLAB to generate a large number of implementations for function evaluation. Several function evaluation methods are considered: polynomial-only (*po*), and table+polynomial of degree two to six (*tp2*-*tp6*). For a given function and any range/precision pair, our MATLAB tool generates ASC code which includes the circuit description, polynomial coefficients and optimized bit-widths. In this fashion, we also obtain minimal bit-widths and the minimal number of polynomial terms for the *po* method. For the *tp* methods, we find the minimal table-size and the coefficient bit-widths for the given range and precision. Fig. 9 shows the structure of our MATLAB tool for algorithmic design space exploration and producing ASC codes for hardware implementations.

The `find_min_degree` function for the *po* method finds the minimal polynomial degree required to meet the output error specification. It starts with a degree one polynomial and finds the minimax polynomial coefficients. For all coefficients, constants, and outputs of operators, which we shall refer as "variables", double precision floating-point is used and the approximation performed. The result is compared to the MATLAB computed value of the function to calculate the approximation error. The polynomial degree is incremented until the desired accuracy is met. The `find_min_seg` function finds the minimal number of segments 2^k needed for a given polynomial degree. It starts with $k = 0$, which is equivalent to *po*, and finds the minimax polynomial coefficients. Again, the approximation of this structure is compared with the MATLAB function evaluation. k is incremented until the maximum error over all segments is lower than the requested error. The `optimize_bw` function performs bit-width optimization on the variables in the data paths. This procedure is discussed in detail in Section VII. The `gen_coeffs` and `gen_table`

functions generate the bit-width optimized polynomial coefficients or coefficient table. Finally, `gen_ASC` generates the ASC code with the circuit description and optimized bit-widths.

Since the IEEE double precision floating-point format used in MATLAB is significantly more accurate than the fixed-point representations we use in this work, we regard double precision as the exact value. In order to verify the correctness of the designs at the algorithmic level, we emulate the function evaluation steps described in Fig. 5 within MATLAB. The emulator is ensured to be a bit-exact version of the actual hardware model, and is used to debug ASC designs. Finite precision effects for fixed-point can be effectively simulated within MATLAB by rounding after each arithmetic operation. For each coefficient and arithmetic operator, we store its fractional bit-width for rounding.

The outputs of the emulator are tested rigorously with a large set of random inputs to confirm that all results are indeed faithfully rounded. For each chunk of ASC code, our tool also generates a report file containing the polynomial degree d used, number of segments for *tp*, fractional bit-widths of the operators, table size, maximum ulp error and the percentage of exactly rounded [11] (accurate to 0.5 ulp) results. 36 fixed-point formats, three functions and six methods are examined. Hence, we explore 648 ASC code segments, generated by our MATLAB tool. The ASC code generation together with the bit-width optimization process described in Section VII, takes approximately ten hours on a dual Intel Xeon 2.6GHz PC with 4GB DDR-SDRAM.

VII. Bit-width Optimization and Error Analysis

It is desirable to minimize the bit-widths for all variables in the data paths, leading to size reductions in tables, and operators such as adders and multipliers. We employ a bit-width minimization scheme which minimizes bit-widths while ensuring that the results meet the the one ulp error bound requirement. We split the problem of minimizing fixed-point bit-widths into two parts: range analysis followed by precision analysis. The two parts are performed entirely within our MATLAB framework, making use of the finite precision hardware emulation models discussed in the previous section. Our function evaluation circuits consist of many different types of operators including adders, barrel shifters, conditionals, dividers, multipliers etc., making the designs complex and difficult to analyze. Hence, a numerical approach is taken to tackle the range and precision minimization problems.

Range analysis involves inspecting the dynamic range and working out the bit-widths of the integer parts. Using insufficient bits for the range can cause overflows or underflows, and excessive bits waste valuable hardware resources. Our range analysis method uses a simulation based approach, where each input of the design is supplied with a large set of random numbers, which ranges over the interval of possible values for the particular input, including the extreme values of that interval. We then record the maximum absolute values for each variable. No rounding is performed, in other words double precision is used throughout.

Let the ith variable be v_i and its maximum absolute value be $v_{i,max}$. By "variable", we refer to coefficients, constants, and outputs of operators in the design. The range bits required for each variable v_i can then be computed with

$$\text{range bits} = \begin{cases} \lceil \log_2(|v_{i,max}|) \rceil + 1 & \text{if} \quad |v_{i,max}| > 1 \\ 1 & \text{if} \quad |v_{i,max}| \leq 1 \end{cases} \tag{4}$$

Given that the number of test samples is large enough, the probabilities of overflows and underflows can be kept arbitrarily low.

Precision analysis involves minimizing the fractional parts of the variables, while respecting the output error criterion. Precision analysis is significantly more challenging and there is a wealth of literature devoted to this topic (e.g. [15], [16]). However, much of the previous work is focused on digital signal processing applications, in which the error analysis criteria (such as signal to noise ratio) are rather different from our needs. In addition, the techniques are rather difficult to implement and slow, making them less amendable to optimize all 648 designs. Hence, we opt for an approach where we keep the factional bit-widths constant.

Let M be the number of variables in the system and the fractional bit-width of the ith variable be b_{v_i}, and the factional bit-widths of the approximation g and the evaluation f be b_g and b_f, respectively (see Fig. 5). Rounding a variable causes a maximum of 0.5 ulp error ($2^{-b_{v_i}-1}$) and truncation causes a maximum of 1 ulp error ($2^{-b_{v_i}}$). Although a rounding circuit requires a small adder, we opt for rounding, since it allows smaller variables than truncation. In order to guarantee faithful rounding, the error ϵ_f at the output f must be

$$\epsilon_f \leq 2^{-b_f}. \tag{5}$$

The error ϵ_f is composed of the following three error terms:

- ϵ_{ap} for approximating g with polynomials;
- ϵ_{vr} for rounding each variable, $\epsilon_{vr} = F(b_{v_0}, \ldots, b_{v_{M-1}})$;
- ϵ_{fr} for rounding the final result f to b_f fractional bits.

The error ϵ_{vr} is effectively the error propagated from the variables in the data paths to the final result. Thus for faithful rounding, one needs to ensure that

$$\epsilon_{ap} + \epsilon_{vr} + \epsilon_{fr} \leq 2^{-b_f}. \tag{6}$$

Rounding f can cause a maximum error of 2^{-b_f-1}, so our requirement can be modified as

$$\epsilon_{ap} + \epsilon_{vr} \leq 2^{-b_f-1}. \tag{7}$$

Making the precisions of the variables b_{v_i} large enough, it is possible to meet this error requirement, since ϵ_{ap} gets arbitrarily small with ϵ_{vr}. The challenge is to keep b_{v_i} as small as possible, while meeting the error requirement in (7). To keep the optimization process simple and fast, we employ uniform (the same) fractional bit-widths for all variables b_{v_i}. We propose a binary search method to find the optimal uniform fractional bit-width b_u. Our definition of "optimal uniform" means that the polynomial degree (in the case of *po*) or number of segments (in the case of *tp*) is the same as `min_degree` or `min_segs` in Fig. 9.

The fractional bit-width of the output of the approximation circuit b_g can be analytically pre-determined by examining the

```
b_u = b_f; // set initial b_u to precision of f
r = 2^6;   // initial search space e.g. 2^6

for i=1:(log2(r)+1)

    r = r/2; // reduce search space by half

    if (m=='po') // for polynomial-only (po)
        my_degree = find_degree(f,i,b_u);
        // compare my_degree with the minimal
        if (my_degree > min_degree)
            b_u = b_u + r;
        else
            b_u = b_u - r;
        end

    elseif (m=='tp') // for table+polynomial (tp)
        my_segs = find_segs(f,i,d,b_u);
        // compare my_segs with the minimal
        if (my_segs > min_segs)
            b_u = b_u + r;
        else
            b_u = b_u - r;
        end
    end

end

b_u = b_u + r;
```

Fig. 10. Structure of our MATLAB tool to find the optimal uniform fractional bit-width b_u.

range reconstruction part. Looking at Fig. 5, the reconstruction of $\sin(x)$ is simply a sign change, hence

$$\text{for } \sin(x), \quad b_g = b_f. \tag{8}$$

For $\log(x)$ reconstruction, there is an addition with a variable. Using one guard bit for the addition,

$$\text{for } \log(x), \quad b_g = b_f + 1. \tag{9}$$

Looking at the reconstruction step of $\sqrt{x}$, g is shifted by $exp1$ (Fig. 5). This means that we need $b_g = \lceil exp + b_f \rceil$ to have enough bits to guarantee b_f fractional bits at the output f. By analyzing the range reduction step, one can see that $exp1$ can be a maximum of $\lceil x.range/2 \rceil$ bits wide. Hence,

$$\text{for } \sqrt{x}, \quad b_g = \lceil x.range/2 \rceil + b_f, \tag{10}$$

where $b_f = b_x = x.prec$.

Fig. 10 shows the structure of our MATLAB tool used for finding the b_u. Using binary search, b_u gradually approaches the optimal in $\log_2(r) + 1$ iterations, where r is the search space. The initial search space r needs to be a power of two and large enough to cover the largest possible b_u.

VIII. Hardware Design Space Exploration and Optimizations

ASC, A Stream Compiler [9], is a C-like programming environment for FPGAs. ASC code makes use of C++ syntax and ASC semantics which allow the user to program on the architecture-level, the arithmetic-level and the gate-level. As a consequence ASC code provides the productivity of high-level hardware design tools and the performance of low-level optimized hardware design. ASC provides types and operators to enable research on custom data representation and arithmetic. Currently supported types are `HWint`, `HWfix` and `HWfloat`, which can store integers, fixed-point numbers and floating-point numbers, respectively. For this paper, we use the `HWfix` type. As a result of this work, function evaluation in ASC is performed with the following declarations and library call:

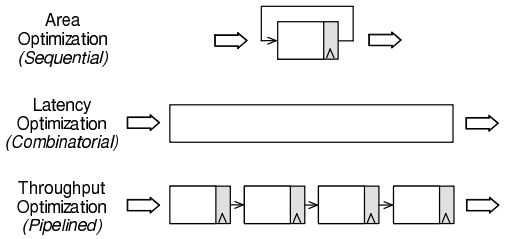

Fig. 11. Principles behind automatic design optimization in ASC. The shaded areas represent flip-flops.

```
HWfix x(TMP, x.range+x.prec, x.prec, sign_mode);
HWfix f(TMP, f.range+f.prec, f.prec, sign_mode);
f = HWsin(x);
```

In order to create an optimized function evaluation library, the MATLAB tool described in Section VI is utilized to generate a large amount of ASC code. This ASC code forms a two-dimensional matrix, which is indexed by range and precision of the argument to the function evaluation call. Each matrix entry consists of a pointer to an ASC function which is called for the particular input x. For instance, for each function we can determine two design selection matrices: for minimal area and for minimal latency. The `HWsin(x)` call indexes into the matrix to find the optimized ASC implementation. The function evaluation code, for example for $\sin(x)$, then indexes into the matrix of function pointers (`HWsin_matrix`) and accesses the correct function based on input range and precision:

```
HWfix &HWsin(HWfix &x){
  return HWsin_matrix[x.range][x.prec](x);
}
```

The design of such matrices are demonstrated in Section IX.

ASC provides an automated mechanism for optimizing designs for user specified metric [9]. The current supported metrics are area, latency and throughput. Fig. 11 illustrates how this is achieved. In area optimization mode, ASC uses sequential arithmetic units, e.g. for multiplication ASC selects an add-accumulate unit. In latency optimization mode, no flip-flops are being inserted and as a consequence the resulting circuit is purely combinational. In throughput optimization mode, all flip-flops that are present in the slices utilized are being used. The resulting circuit is balanced (scheduled) by using FIFO buffers in between the arithmetic units.

All together, the 2000 lines of MATLAB code generate 648 hardware designs targeting FPGAs, which result in 300,000 lines of ASC code. We also generate a number of additional designs to examine the area cost of range reduction, which is discussed in Section IX. For each design, ASC generates three designs which are optimized for area, latency and throughput. The result is a huge experimentation space of over 2000 FPGA designs. These are placed-and-routed on the recently-released

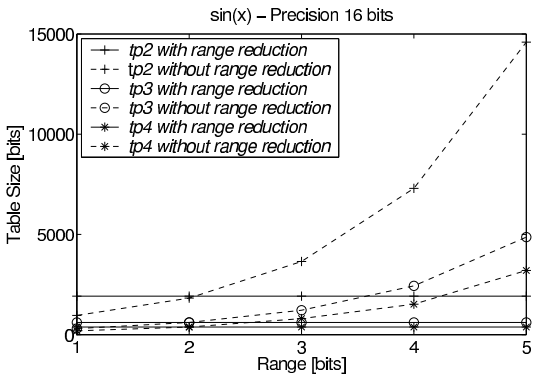

Fig. 12. Table size comparison when evaluating $\sin(x)$ at precision of 16 bits with range reduction and without range reduction.

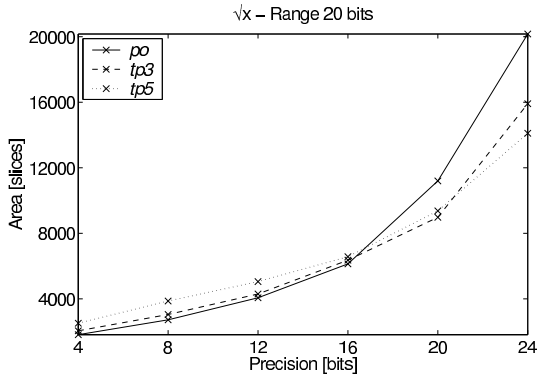

Fig. 13. Area usage of different methods at various precisions for evaluating $\sqrt{x}$ with a range of 20 bits.

Xilinx Virtex-4 XC4VLX200-11 FPGA, which is the largest device of the Xilinx Virtex-4 LX family. The designs are synthesized with ASC and placed-and-routed with Xilinx ISE 6.3, resulting in over 100 million ASIC equivalent gates. This work flow which is fully automated with a single "makefile", takes a weeks' time on two dual Intel Xeon 2.6GHz PCs with 4GB DDR-SDRAM. The makefile accepts design space exploration parameters including range/precision sets, functions, approximation methods and metric optimizations. The final output is a report file containing area, latency and throughput results for the user specified parameters.

IX. Results

In this section, we present device independent and placed-and-routed FPGA results. The device independent results are obtained using our MATLAB tool at the algorithmic design space exploration stage, showing table sizes and bit-widths of the variables. The placed-and-routed results are obtained using ASC and Xilinx ISE on a Xilinx Virtex-4 XC4VLX200-11 FPGA device.

A. Device Independent Results with MATLAB

Before mapping designs into actual hardware devices, it is interesting to explore the tradeoffs at the algorithmic level. The plots in Fig. 14 show the table size and uniform fractional width b_u variations at different ranges and precisions using *tp3*. We observe that for all three functions, the table size grows with precision. $\sqrt{x}$ has the largest table size requirement, followed by $\sin(x)$ and $\log(x)$. This follows from the discussions in Section IV: more resources are needed for functions with a large approximation interval and a large first derivative. The table size increases with both range and precision for $\sqrt{x}$. This is due to the accuracy of approximation g being dependent on both range and precision as seen in (10). From (8) and (9), g is independent of the range for $\sin(x)$ and $\log(x)$. Hence for $\log(x)$, we see no change in table size with range. However, slight increase can be seen for $\sin(x)$. This is because the complexity of the modulus operation which incorporates a divider in the range reduction circuit of $\sin(x)$, increases with the input range.

The size of b_u gives us an indication of the operator complexities in the design. We see that b_u increases in a linear manner with range and precision except for $\log(x)$, where it stays pretty much constant with range. As noted earlier, the complexity of the range reduction circuit of $\sin(x)$ increases with range. For $\sqrt{x}$, the accuracy requirement of the approximation circuit grows with range. Hence, the increase in b_u for the two functions. Looking at the three functions, the bit-width requirement of $\log(x)$ is low, whereas $\sin(x)$ and $\sqrt{x}$ are both high. Finally, the plots Fig. 14 give us an indication that $\sqrt{x}$ takes the most area, followed by $\sin(x)$ and $\log(x)$.

Fig. 12 shows a comparison in table size as a function of range, when evaluating $\sin(x)$ at precision of 16 bits with range reduction and without range reduction for three *tp* methods. Note that the term range reduction is used to also include range reconstruction. When the function is evaluated without range reduction, the whole input interval is approximated without the use of a range reduction step. As one would expect, the table size stays constant with range when range reduction is used. However, when range reduction is not used, the table size increases exponentially with the range, since each additional bit in the range doubles the interval of approximation. We note that when the range is small (e.g. less than two bits for *tp2*), it is more sensible to skip range reduction due to the smaller table size.

B. Placed-and-Routed Results on FPGA

We summarize the results of the 2000 FPGA implementations obtained with the ASC system discussed in Section VIII. One dimension of the design space is technology mapping on the FPGA side. In addition to slices, the Virtex-4 FPGA contains embedded RAMS and multiply-and-add blocks. However, in this work, we decide to use slices only to make the comparisons easier and fairer. To implement the coefficient tables, the 4-input LUTs are used together with logic minimization [17]. Instead of using the 4-input LUTs directly as memory (known as distributed RAM), this approach can lead to smaller and faster tables for the designs used in this work.

Fig. 13 shows the area requirements of *po*, *tp3* and *tp5* at various precisions for evaluating $\sqrt{x}$ with a range of 20 bits. We recognize that this figure is analogous to the example used in Fig. 1. For these particular set of parameters, for precisions

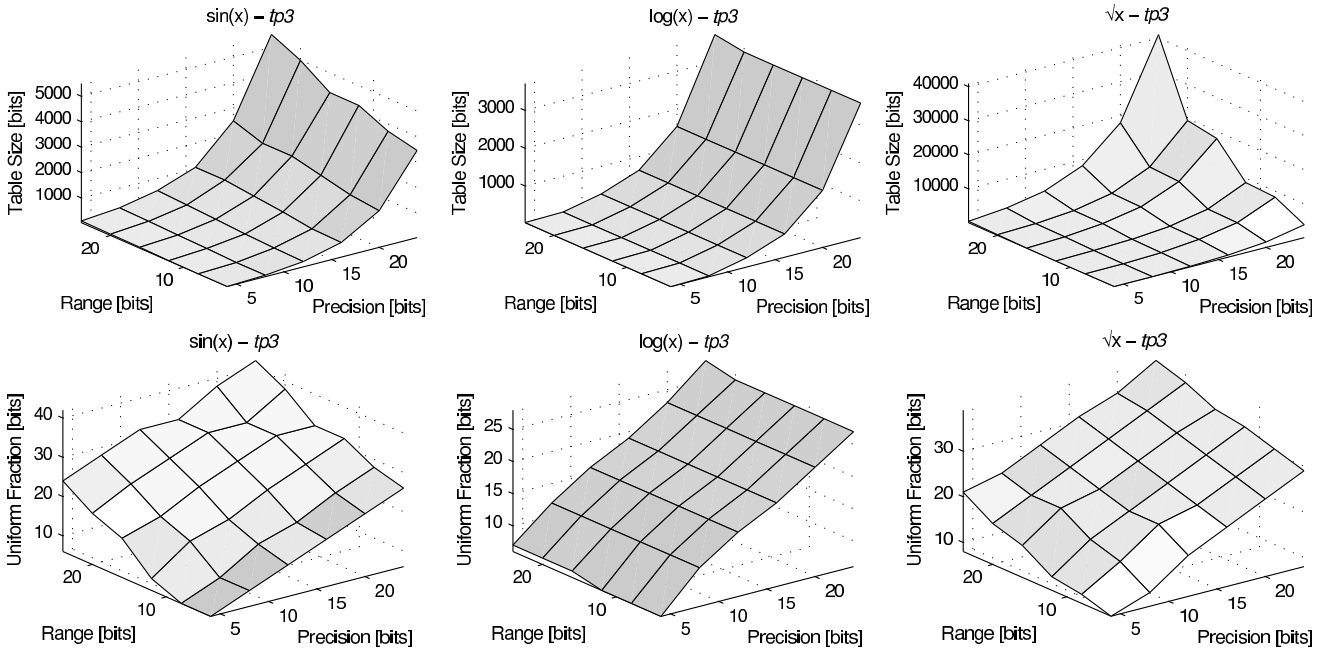

Fig. 14. **Device independent results with MATLAB:** table size and uniform fractional bit-width b_u variations at different ranges/precisions using *tp3*.

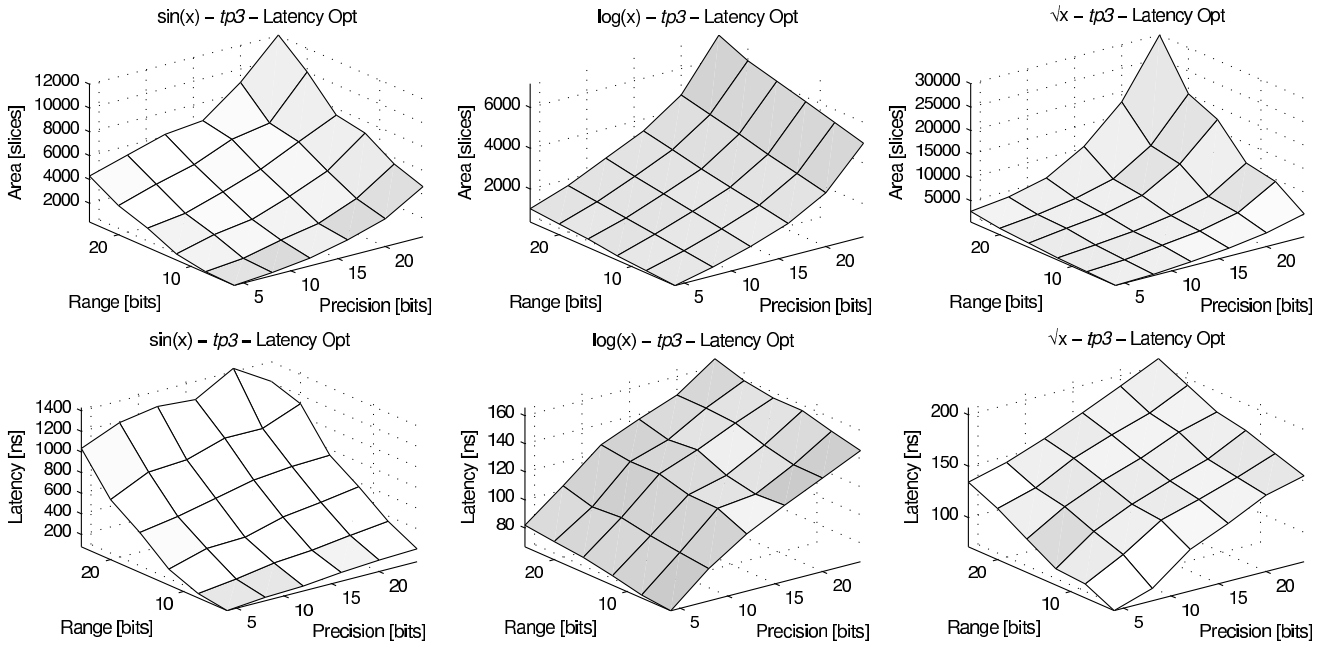

Fig. 15. **Placed-and-routed results on FPGA:** area and latency variations at different ranges/precisions using *tp3* with latency optimization.

less than 16 bits, *po* results in the minimal area. *tp3* gives the least area between 16 and 20 bits, and *tp5* provides the least area above 20 bits.

Fig. 15 shows the area and latency variations for various range/precision combinations using *tp3*. Latency optimization is chosen to illustrate these design spaces, since combinatorial circuits best reflect the complexity of designs. Looking at the two figures, we see a remarkable consistency to the device independent results in Fig. 14, suggesting that our approach could be applied across different device technologies.

From the area results in Fig. 15, we observe that $\sqrt{x}$ requires the most area, followed by $\sin(x)$ and $\log(x)$. This can be explained by the table size requirements shown in Fig. 14: large tables lead to large utilizations of 4-input LUTs. Whereas the table size for $\sin(x)$ stays relatively constant with range in Fig. 14, the area usage is actually increasing. This is probably due to the presence of divider in the range reduction step, whose area requirement increases with its operand size. The latency results area related to the uniform fractional bit-width b_u in Fig. 14. Since b_u dictates the size of the operands such as adders, dividers and multipliers.

Fig. 16 and 17 highlight the area cost of range reduction for

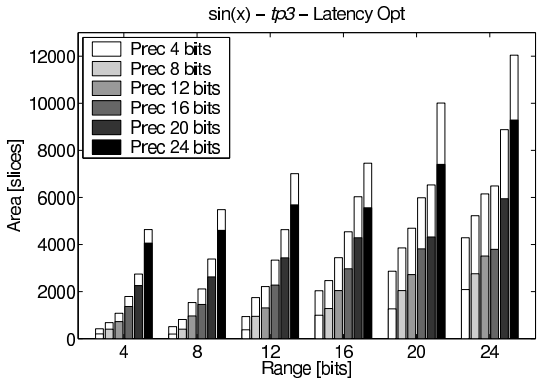

Fig. 16. Area cost of range reduction (upper part) for $\sin(x)$ using *tp3* and latency optimization.

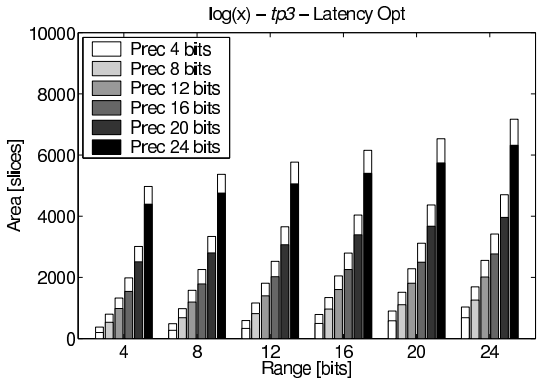

Fig. 17. Area cost of range reduction (upper part) for $\log(x)$ using *tp3* and latency optimization.

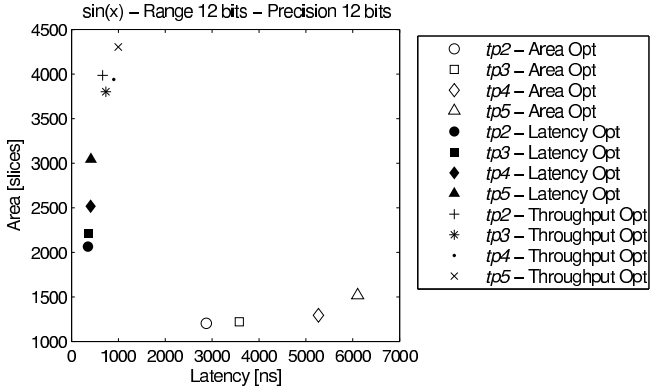

Fig. 18. Pareto-optimal points in the area-latency space for 12-bit range and 12-bit precision evaluation to $\sin(x)$.

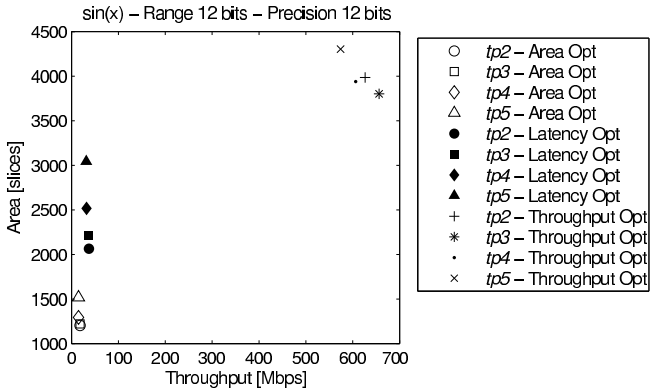

Fig. 19. Pareto-optimal points in the area-throughput space for 12-bit range and 12-bit precision evaluation to $\sin(x)$.

$\sin(x)$ and $\log(x)$, with the approximation circuit implemented using *tp3*. The lower part of the bars shows the slices used for function approximation, and the small upper part shows the slices used for range reduction. For both functions, it can be seen that the cost of range reduction grows with range and precision. This is mostly due to the modulus incorporated in $\sin(x)$, and the barrel shifter and multiplier in $\log(x)$, which are all affected by the operator size. The range reduction cost for $\sin(x)$ is considerably higher than $\log(x)$, because the modulus operator contains a division. On average, the percentage area used by range reduction for $\sin(x)$ and $\log(x)$ are 37% and 23%, respectively. The behavior of $\sqrt{x}$ is found to be similar to $\log(x)$, due to their resemblance in their range reduction circuits.

The scatter plots in Fig. 18 and 19 highlight the Pareto-optimal [17] points in the area-latency and area-throughput space. The evaluation of $\sin(x)$ with 12-bit range and 12-bit precision is chosen as an example. Assorted *tp* methods are shown, performed with area, latency and throughput optimizations. As expected, designs optimized for a particular metric result in best performance in its own metric. With the aid of such plots, one can decide rapidly what methods to use for meeting specific requirements in area, latency or throughput. Focusing on the latency optimized results in Fig. 18, *tp* designs with lower polynomial degrees are always going to the faster due to their shallower multiply-and-add tree illustrated in Fig. 8. In terms of area, low polynomial degree designs are generally smaller for the same reason. However, table size grows with the precision required, making low polynomial degree designs potentially larger, as demonstrated in Fig. 13.

As proposed in Section VIII, Fig. 20 sums up the most interesting results in two matrices, which show the Pareto-optimal solutions for different range/precision pairs. Although many more matrices can be generated for different metric and optimization combinations, we choose these two matrices for illustration purposes of our approach. The first matrix shows designs that result in minimal area with area optimization, and the second matrix shows designs that result in minimal latency with latency optimization. For instance, from the first matrix, the dashed box tells us that for a $\sin(x)$ design with 12-bit range and 12-bit precision, the smallest implementation would be *tp2* with a uniform fractional bit-width of 20 bits and a table size of 504 bits. In essence, these matrices tell us, for each combination of range and precision, which method to use for the three functions to get the minimal metric.

C. Performance of the Units and their Usage

Recent processors such as those based on the IA-64 architecture can evaluate functions in between 50 and 70 clock cycles [18]. Considering a typical processor clock speed of

Minimal Area (Optimized for Area)

Range [bits] \ Precision [bits]	4	8	12	16	20	24
24	**sin**: *po2*, 24, 76 **log**: *po2*, 7, 27 **sqr**: *tp2*, 18, 912	**sin**: *tp2*, 28, 348 **log**: *tp2*, 12, 78 **sqr**: *tp4*, 22, 920	**sin**: *tp2*, 32, 792 **log**: *tp2*, 15, 384 **sqr**: *tp6*, 27, 784	**sin**: *tp3*, 32, 1056 **log**: *tp3*, 19, 640 **sqr**: *tp5*, 33, 1632	**sin**: *po6*, 40, 288 **log**: *tp4*, 24, 1000 **sqr**: *tp6*, 35, 2016	**sin**: *tp5*, 42, 1032 **log**: *tp3*, 28, 3712 **sqr**: *tp6*, 39, 4480
20	**sin**: *po2*, 19, 61 **log**: *po2*, 7, 27 **sqr**: *tp2*, 18, 456	**sin**: *tp2*, 24, 300 **log**: *tp2*, 12, 78 **sqr**: *tp4*, 20, 420	**sin**: *tp2*, 28, 696 **log**: *tp2*, 15, 384 **sqr**: *tp4*, 24, 1000	**sin**: *tp3*, 32, 1056 **log**: *tp3*, 19, 640 **sqr**: *tp5*, 28, 1392	**sin**: *tp5*, 32, 792 **log**: *tp4*, 24, 1000 **sqr**: *tp6*, 32, 1848	**sin**: *tp5*, 38, 936 **log**: *tp5*, 28, 696 **sqr**: *tp6*, 37, 4256
16	**sin**: *po2*, 16, 52 **log**: *po2*, 7, 27 **sqr**: *tp3*, 15, 128	**sin**: *tp2*, 19, 240 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 18, 912	**sin**: *tp2*, 24, 600 **log**: *tp2*, 15, 384 **sqr**: *tp4*, 22, 920	**sin**: *po5*, 28, 174 **log**: *tp3*, 19, 640 **sqr**: *tp6*, 27, 784	**sin**: *po6*, 32, 232 **log**: *tp4*, 24, 1000 **sqr**: *tp5*, 33, 1632	**sin**: *tp5*, 32, 792 **log**: *tp5*, 28, 696 **sqr**: *tp6*, 35, 2016
12	**sin**: *po2*, 9, 31 **log**: *po2*, 7, 27 **sqr**: *tp2*, 12, 156	**sin**: *tp2*, 16, 204 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 18, 456	**sin**: *tp2*, 20, 504 **log**: *tp2*, 15, 384 **sqr**: *tp4*, 20, 420	**sin**: *tp3*, 24, 800 **log**: *tp3*, 19, 640 **sqr**: *tp4*, 24, 1000	**sin**: *po6*, 28, 204 **log**: *tp4*, 24, 1000 **sqr**: *tp6*, 29, 840	**sin**: *tp5*, 32, 792 **log**: *tp5*, 28, 696 **sqr**: *tp6*, 32, 1848
8	**sin**: *po2*, 6, 22 **log**: *tp2*, 6, 21 **sqr**: *tp2*, 10, 66	**sin**: *tp2*, 10, 132 **log**: *tp2*, 11, 72 **sqr**: *tp3*, 15, 128	**sin**: *tp2*, 16, 408 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 18, 912	**sin**: *tp4*, 19, 400 **log**: *tp3*, 19, 640 **sqr**: *tp4*, 22, 920	**sin**: *tp5*, 24, 600 **log**: *tp4*, 24, 1000 **sqr**: *tp6*, 27, 784	**sin**: *tp5*, 28, 696 **log**: *tp5*, 28, 696 **sqr**: *tp5*, 28, 1632
4	**sin**: *po2*, 6, 22 **log**: *tp2*, 6, 21 **sqr**: *po2*, 7, 25	**sin**: *tp2*, 10, 132 **log**: *tp2*, 11, 72 **sqr**: *tp2*, 12, 156	**sin**: *tp3*, 14, 240 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 18, 456	**sin**: *tp4*, 18, 380 **log**: *tp3*, 19, 640 **sqr**: *tp4*, 20, 420	**sin**: *tp5*, 22, 552 **log**: *tp5*, 24, 600 **sqr**: *tp4*, 24, 1000	**sin**: *tp4*, 27, 1120 **log**: *tp5*, 28, 696 **sqr**: *tp6*, 29, 840

Minimal Latency (Optimized for Latency)

Range [bits] \ Precision [bits]	4	8	12	16	20	24
24	**sin**: *tp2*, 24, 78 **log**: *po2*, 8, 30 **sqr**: *tp2*, 18, 912	**sin**: *tp2*, 28, 348 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 24, 2400	**sin**: *tp2*, 32, 792 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 26,10368	**sin**: *tp2*, 32, 3168 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 30,23808	**sin**: *tp2*, 40, 7872 **log**: *tp2*, 24, 4800 **sqr**: *tp3*, 34,17920	**sin**: *tp3*, 42, 5504 **log**: *tp2*, 28,11136 **sqr**: *tp4*, 39,12800
20	**sin**: *po2*, 19, 61 **log**: *po2*, 7, 27 **sqr**: *tp2*, 18, 456	**sin**: *tp2*, 24, 300 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 20, 2016	**sin**: *tp2*, 28, 696 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 24, 4800	**sin**: *tp2*, 32, 3168 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 32,12288	**sin**: *tp2*, 32, 6336 **log**: *tp2*, 24, 4800 **sqr**: *tp3*, 33, 8704	**sin**: *tp3*, 38, 4992 **log**: *tp2*, 27,10752 **sqr**: *tp3*, 37,19456
16	**sin**: *tp2*, 16, 54 **log**: *po2*, 7, 27 **sqr**: *tp2*, 17, 324	**sin**: *tp2*, 19, 240 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 18, 912	**sin**: *tp2*, 24, 600 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 24, 2400	**sin**: *tp2*, 28, 2784 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 26,10368	**sin**: *tp2*, 32, 6336 **log**: *tp2*, 24, 4800 **sqr**: *tp2*, 30,23808	**sin**: *tp3*, 32, 4224 **log**: *tp2*, 27,10752 **sqr**: *tp3*, 34,17920
12	**sin**: *po2*, 9, 31 **log**: *po2*, 7, 27 **sqr**: *tp2*, 12, 156	**sin**: *tp2*, 16, 204 **log**: *tp2*, 12, 78 **sqr**: *tp2*, 18, 456	**sin**: *tp2*, 20, 504 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 20, 2016	**sin**: *tp2*, 24, 2400 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 24, 4800	**sin**: *tp2*, 28, 5568 **log**: *tp2*, 24, 4800 **sqr**: *tp2*, 31,12288	**sin**: *tp2*, 32,12672 **log**: *tp2*, 27,10752 **sqr**: *tp3*, 33, 8704
8	**sin**: *po2*, 6, 22 **log**: *po2*, 7, 27 **sqr**: *tp2*, 7, 27	**sin**: *tp2*, 10, 132 **log**: *tp2*, 11, 72 **sqr**: *tp2*, 17, 324	**sin**: *tp2*, 16, 408 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 18, 912	**sin**: *tp2*, 19, 1920 **log**: *tp2*, 21, 1056 **sqr**: *tp2*, 24, 2400	**sin**: *tp2*, 24, 4800 **log**: *tp2*, 24, 4800 **sqr**: *tp2*, 26,10368	**sin**: *tp2*, 28,11136 **log**: *tp2*, 27,10752 **sqr**: *tp2*, 30,23808
4	**sin**: *po2*, 6, 22 **log**: *po2*, 7, 27 **sqr**: *po2*, 7, 25	**sin**: *tp2*, 10, 132 **log**: *tp2*, 11, 72 **sqr**: *tp2*, 12, 156	**sin**: *tp2*, 15, 384 **log**: *tp2*, 15, 384 **sqr**: *tp2*, 18, 456	**sin**: *tp2*, 19, 1920 **log**: *tp2*, 17, 1056 **sqr**: *tp2*, 20, 2016	**sin**: *tp2*, 23, 4608 **log**: *tp2*, 24, 4800 **sqr**: *tp2*, 24, 4800	**sin**: *tp2*, 28,11136 **log**: *tp2*, 27,10752 **sqr**: *tp2*, 31,12288

Fig. 20. Area (with area optimization) and latency (with latency optimization) matrices showing for each range/precision combination, the design with minimal area and latency. For each entry, the optimal method, uniform fractional bit-width and table size in bits are shown. The number after *po* indicates the polynomial degree used. For instance, the dashed box tells us that for a $\sin(x)$ design with 12-bit range and 12-bit precision, the smallest implementation would be *tp2* with a uniform fractional bit-width of 20 bits and a table size of 504 bits.

3GHz, this means that a result can be produced in around 20ns. The automated throughput optimized designs in this work are fully pipelined and have a clock speed of around 100MHz (much higher clock speeds could be achieved with manual pipelining), meaning that we can produce a result every 10ns, which is a speed-up of a factor of two over a 3GHz processor. Since one function evaluation unit does not take much space on an FPGA, we could have multiple units running in parallel, potentially resulting in orders of magnitude speed-up.

The proposed method is developed to produce, for a given function, metric, range and precision, an optimal hardware function evaluation unit. The results can be arranged in the form of matrices as shown in Fig. 20. Our method can be seen as a step in the hardware optimization process, after the range and precision have been determined by application developers or by other methods [16].

X. Conclusions

A methodology and an automated function evaluation unit generation, for given function and a set of user requirements with custom range and precision values have been presented. The result is an optimized fixed-point function evaluation generator library for hardware designs. Our approach has been demonstrated with three elementary functions $\sin(x)$, $\log(x)$ and $\sqrt{x}$ for 36 range and precision combinations between 8 and 48 bits. MATLAB is used for algorithmic design space

exploration and ASC code generation, while ASC is used to perform hardware design space exploration targeting FPGAs.

The degrees of freedom including applicability of range reduction, approximation method selection, and hardware optimization have been discussed. Bit-width optimization techniques for minimizing both range and precision have been proposed, based on a binary search technique. We have examined various device independent and device specific results, covering a vast design space of over 2000 designs, equivalent to 100 million ASIC gates. We have shown two matrices showing for each range/precision combination, which approximation method to use for minimal area and latency. We conclude that the automation of optimized hardware function evaluation is already within reach.

ACKNOWLEDGMENTS

The authors thank Ray C.C. Cheung, David Pearce and the anonymous reviewers for their assistance. The support of the Jet Propulsion Laboratory, Xilinx Inc., and the U.K. Engineering and Physical Sciences Research Council (Grant number GR/N 66599, GR/R 55931 and GR/R 31409) is gratefully acknowledged.

REFERENCES

[1] E. O'Grady and C. Wang, "Performance limitations in parallel processor simulations," *Trans. Society for Computer Simulation*, vol. 4, pp. 311–330, 1987.
[2] J. Muller, *Elementary Functions: Algorithms and Implementation*. Birkhauser Verlag AG, 1997.
[3] O. Mencer and W. Luk, "Parameterized high throughput function evaluation for FPGAs," *J. VLSI Signal Processing*, vol. 36, no. 1, pp. 17–25, 2004.
[4] J.E. Stine and M.J. Schulte, "The symmetric table addition method for accurate function approximation," *J. VLSI Signal Processing*, vol. 32, no. 2, pp. 167–177, 1999.
[5] R. Andraka, "A survey of CORDIC algorithms for FPGA based computers," in *Proc. ACM/SIGDA Int'l Symp. Field-Programmable Gate Arrays*, 1998, pp. 191–200.
[6] A. Peymandoust and G. De Micheli, "Application of symbolic computer algebra in high-level data-flow synthesis," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 9, pp. 1154–1165, 2003.
[7] C. Maxfield, *The Design Warrior's Guide to FPGAs*. Newnes, 2004.
[8] *Virtex-4 Family Overview*, Xilinx Inc., 2004, http://www.xilinx.com.
[9] O. Mencer, D. Pearce, L. Howes, and W. Luk, "Design space exploration with A Stream Compiler," in *Proc. IEEE Int'l Conf. Field-Programmable Technology*, 2003, pp. 270–277.
[10] D. Das Sarma and D. Matula, "Faithful bipartite ROM reciprocal tables," in *Proc. IEEE Symp. Computer Arithmetic*, 1995, pp. 17–28.
[11] M.J. Schulte and E.E. Swartzlander Jr, "Hardware designs for exactly rounded elementary functions," *IEEE Trans. Computers*, vol. 43, no. 8, pp. 964–973, 1994.
[12] J. Walther, "A unified algorithm for elementary functions," in *Proc. AFIPS Spring Joint Computer Conf.*, 1971, pp. 379–385.
[13] D. Lee, A. Abdul Gaffar, O. Mencer, and W. Luk, "Adaptive range reduction for hardware function evaluation," in *Proc. IEEE Int'l Conf. Field-Programmable Technology*, 2004, pp. 169–176.
[14] I. Koren and O. Zinaty, "Evaluating elementary functions in a numerical coprocessor based on rational approximations," *IEEE Trans. Computers*, vol. 39, no. 8, pp. 1030–1037, 1990.
[15] G. Constantinides, P. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 10, pp. 1432–1442, 2003.
[16] K. Kum and W. Sung, "Combined word-length optimization and high-level synthesis of digital signal processing systems," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 921–930, 2001.
[17] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
[18] J. Harrison, T. Kubaska, S. Story, and P. Tang, "The computation of transcendental functions on the IA-64 architecture," *Intel Technology Journal*, vol. Q4, 1999.

Dong-U Lee (S'01-M'05) received the BEng degree in information systems engineering and the PhD degree in computing, both from Imperial College London in 2001 and 2004, respectively. He is currently a postdoctoral researcher at the Electrical Engineering Department, University of California, Los Angeles (UCLA), where he is working on channel codes and symbol timing synchronization for deep-space communications with the Jet Propulsion Laboratory, NASA. His research interests include computer arithmetic, communications, design automation, reconfigurable computing and video image processing. He is a member of the IEEE.

Altaf Abdul Gaffar received the BEng degree in information systems engineering from Imperial College London in 2000 and is completing the PhD degree in computing at the same university. He is also currently working as a research assistant at the Electrical and Electronic Engineering Department at Imperial College London. His research interests include bit-width optimization for floating-point and fixed-point arithmetic, and high-level power estimation and optimization techniques. He is a member of the IEEE.

Oskar Mencer received his PhD in electrical engineering from Stanford University. He founded MAXELER Technologies in 2003 after three years as Member of Technical Staff in the Computing Sciences Research Center at Bell Labs. He is currently a member of academic staff in the Department of Computing at Imperial College London. His research interests span computer architecture, computer arithmetic, VLSI microarchitecture, VLSI CAD, and reconfigurable (custom) computing. He is a member of the IEEE.

Wayne Luk (S'85-M'89) received the MA, MSc, and PhD degrees in engineering and computer science from the University of Oxford. He is a member of academic staff in Department of Computing, Imperial College London and leads the Custom Computing Group there. His research interests include theory and practice of customizing hardware and software for specific application domains, such as graphics and image processing, multimedia, and communications. 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 member of the IEEE.