Automating Custom-Precision Function Evaluation for Embedded Processors

Ray C.C. Cheung
Department of Computing
Imperial College London
London, United Kingdom
r.cheung@imperial.ac.uk

Dong-U Lee
EE Department
University of California
Los Angeles, USA
dongu@icsl.ucla.edu

Oskar Mencer
Department of Computing
Imperial College London
London, United Kingdom
o.mencer@imperial.ac.uk

ABSTRACT

Due to resource and power constraints, embedded processors often cannot afford dedicated floating-point units. For instance, the IBM PowerPC processor embedded in Xilinx Virtex-II Pro FPGAs only supports emulated floating-point arithmetic, which leads to slow operation when floating-point arithmetic is desired. This paper presents a customizable mathematical library using fixed-point arithmetic for elementary function evaluation. We approximate functions via polynomial or rational approximations depending on the user-defined accuracy requirements. The data representation for the inputs and outputs are compatible with IEEE single-precision and double-precision floating-point formats. Results show that our 32-bit polynomial method achieves over 80 times speedup over the single-precision mathematical library from Xilinx, while our 64-bit polynomial method achieves over 30 times speedup.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: Real-time and embedded systems; D.3.4 [**Programming Languages**]: Processors—*code generation, optimization.*

General Terms

Measurement, Performance, Design.

Keywords

Embedded systems, reconfigurable computing, function evaluation, fixed-point arithmetic.

1. INTRODUCTION

The evaluation of elementary functions is often the performance bottleneck of many compute-bound applications [15]. Examples of these functions include logarithm $\log(x)$ and square root $\sqrt{x}$. Evaluating such functions efficiently while meeting the precision requirements is particulary important for embedded applications, where stringent resource and power constraints are enforced.

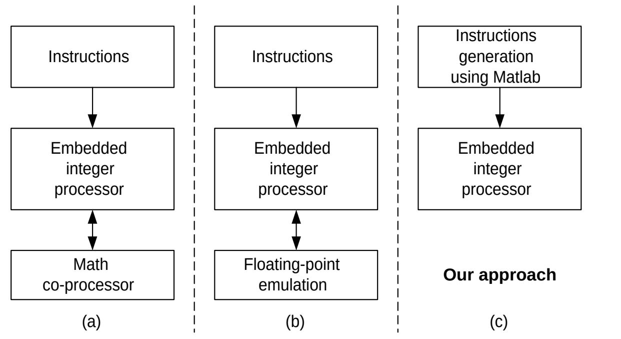

Instructions
Embedded integer processor
Math co-processor
(a)
Instructions
Embedded integer processor
Floating-point emulation
(b)
Instructions generation using Matlab
Embedded integer processor
Our approach
(c)

Figure 1: Overview of current and the proposed embedded processors. Current approach includes (a) using co-processor and (b) using floating-point emulation.

Advanced FPGAs enable the development of configurable SoC systems and high-speed function evaluation units that are customized to particular applications. As shown in Figure 1(a), in embedded systems, the integer processor is usually incorporated with one or more dedicated coprocessors such as a math coprocessor for fast function evaluation, which results in a tradeoff between area, cost and performance. Figure 1(b) illustrates the emulated floating-point mathematical library from Xilinx [5]. In this approach, floating-point arithmetic is emulated using integer operations only without the use of a coprocessor [1]. Performance degradation and code space consumption are the two major problems for using this approach.

In this paper, we propose an Integer Mathematical Generation tool, **IMGen**, which makes use of optimized fixed-point (integer) arithmetic for internal computations. IEEE single and double precision floating-point formats are used for both the input and output formats such that internal computation is transparent to the users. A design generator is used to automatically select the best polynomial/rational approximation for internal computations and the degree of computation for a given error tolerance.

[1]Recently, Xilinx has released the Virtex-4 FX FPGA which has an Auxiliary Processor Unit (APU) [1] that can connect the math coprocessor using FPGA fabrics. In this work, we compare the designs without using this math coprocessor.

Table 1: Instruction latency comparison of different processors. (* using floating-point co-processor.)

	ARM 7	PowerPC 750	MicroBlaze	MIPS32 24k
add	1	1	1	4
multi	3	5	3	4
fmulti	4*	4*	6*	4
idiv	n/a	19	34	32

The main contributions of this paper are:

- **IMGen**, a customizable library for floating-point function evaluation based on the integer instruction set found in embedded processors.
- automatic code generation and optimization within IMGen using Matlab to customize precision, and trade offs involving code space, performance and accuracy.
- an evaluation of the proposed approach by automatically selecting polynomial or rational approximation for a given function, accuracy requirement and execution time for embedded integer processors.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the integer instruction set of the IBM PowerPC405. Section 4 presents the methodology for enabling the automatic library generation and optimizing the code generation. Section 5 provides error analysis for our function evaluation library. Section 6 describes the optimizations we perform for code generation and code space minimization. Section 7 discusses results, and Section 8 provides conclusions and future work.

2. RELATED WORK

We consider an elementary function $f(x)$, where x and $f(x)$ have a given range $[a, b]$. The evaluation of $f(x)$ usually consists of three steps [19]: (1) range reduction [8] which reduces x over the interval $[a, b]$ to a more convenient x' over a smaller interval $[a', b']$, (2) function approximation on the reduced interval, and (3) range reconstruction which expands the final result back to the original result range.

Function evaluation for both hardware designs [23] and software designs [14] has been well presented in the literature. Lee et al. [15] explore the effects of using different input ranges and precisions on FPGA area and speed. Mencer et al. [18] study pipelined function evaluation using table lookup, CORDIC , rational and polynomial approximations. Other efforts include using small multipliers [9], IBM RISC System/6000 [17], AMD K5 processor [16], Intel IA-64 [13] in recent years.

Paul et al. [22] discuss if dedicated instruction sets for elementary function evaluation should be incorporated into computer systems. On the other hand, much research has been focused on accuracy, such as multiple-precision for function evaluation [7], IEEE floating-point standard mathematical library [10], quad-double precision [11] and quadruple precision [6] arithmetic.

Recent work [12] develops a math library for the Intel XScale^{TM} architecture that has an efficient software implementation of the basic operations and math library routines. Moreover, code optimization for Digital Signal Processors (DSP) is also important. Another recent work [20] shows that optimized C functions can achieve up to 300% performance gain. In this paper, we achieve performance improvements by using the generated Integer Mathematical Library, **IMGen**, and an integer processor without any additional hardware cost. Our proposed method is easily applicable to any embedded integer processor such as the Xilinx MicroBlaze [3], ARM processor in Altera Excalibur [2] and DSPs.

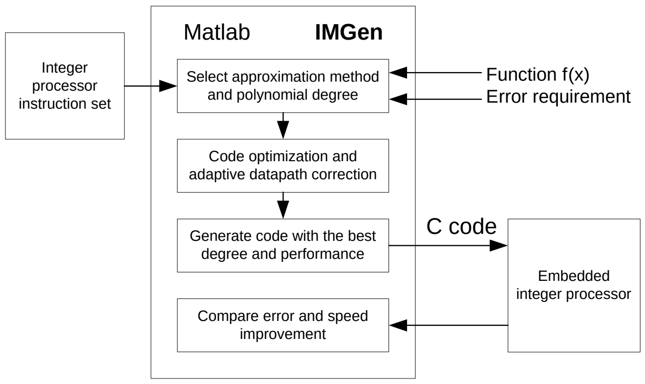

Matlab
IMGen
Integer processor instruction set
Select approximation method and polynomial degree
Function f(x)
Error requirement
Code optimization and adaptive datapath correction
Generate code with the best degree and performance
C code
Embedded integer processor
Compare error and speed improvement

Figure 2: Design tool flow using Matlab to generate the library C source code for the library. The user can specify the require accuracy/error requirement for the library.

3. INTEGER INSTRUCTION SETS

Instruction processors play an important role in embedded systems. We classify these processors based on their datapath width such as 8-bit, 16-bit, etc. The automated library generation framework proposed in this paper is applicable to all integer processors. To demonstrate our customizable precision-based function evaluation method, **IMGen**, we have selected the IBM PowerPC405 from Xilinx as a platform. The PowerPC405 [5] is a 32-bit implementation of the PowerPC embedded-environment architecture that is derived from the PowerPC architecture. It also has a fixed-point execution unit that is used for 32-bit computation and has thirty-two 32-bit general purpose registers. The processor has 16KB 2-way set associative instruction-cache and data-cache, respectively. In this work, we set the clock speed of the processor, the processor local bus and the bus components to 100MHz.

Industrial integer processors include ARM, MIPS, PowerPC and the Xilinx embedded software processor MicroBlaze. Table 1 shows a latency comparison of different architectures. We include some important instructions such as multiplication (*multi*) and integer division (*idiv*) in this comparison. The design method we present in this paper enables code generation for all these integer processors.

4. AUTOMATION METHODOLOGY

This section introduces our approach to automating the customization of precision based function evaluation for integer processors. The method we propose is technology-independent, and we use the embedded PowerPC processor as a specific example to demonstrate the flexibility and performance gain of this method.

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

```
// Range Reduction
input.sng_as_flt = x;
exp = input.sng_as_fld.exp - 126;
ix = fp2int(input); // y = ix;

// Evaluation Method
// f(y) where y = [0.5, 1)
// e.g. degree-3 polynomial
$f1 = ((c_3 \times y + c_2) \times y + c_1) \times y + c_0$;

// Range Reconstruction
s1 = range(exp); // find $\exp \times \log(2)$
f1 = (f1 >> overflow) + s1;
output = int2fp(f1);
output.sng_as_fld.exp += overflow;
```

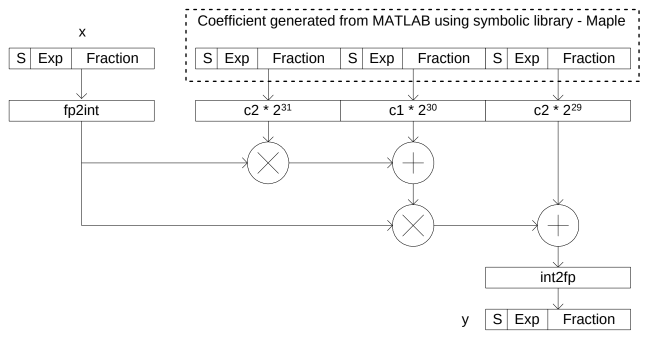
Evaluating $f(x) = \sqrt{x}$

```
// Range Reduction
input.sng_as_flt = x;
exp = input.sng_as_fld.exp - 126;
ix = fp2int(input);
ix = (exp[0])? ix >> 1 : ix; // y = ix;

// Evaluation Method
// f(y) where y = [0.25, 1)
// e.g. degree-2 rational
$f1 = ((c_2 \times y + c_1) \times y + c_0)/((d_2 \times y + d_1) \times y + d_0)$;

// Range Reconstruction
exp = (exp[0])? exp+1 : exp;
correction = exp >> 1;
output.sng_as_fld.exp += exp - correction;
```

Figure 3: Design methods using polynomial and rational approximations. For example, $\log(x)$ uses degree-3 polynomial approximation and $\sqrt{x}$ uses degree-2 rational approximation.



x
Coefficient generated from MATLAB using symbolic library - Maple
S
Exp
Fraction
fp2int
c2 * 2^31
c1 * 2^30
c2 * 2^29
int2fp
y

Figure 4: Maple generates the polynomial coefficients and the IMGen generator adjusts the datapath that can maximize the accuracy. x is input, $y = f(x)$ is output.

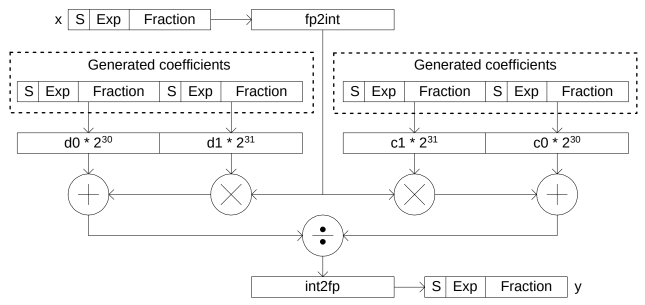


Figure 5: Maple generates the rational coefficients and the IMGen generator adjusts the datapath that can maximize the accuracy. x is input, $y = f(x)$ is output.

4.1 Overview

The general idea of this work is user-transparent fixed-point computation that provides an efficient floating-point function library without any additional hardware cost. A library generator shown in Figure 2 is developed for constructing **IMGen** which targets various integer processors. The Matlab based code generator, which has direct access to the Maple symbolic library, inputs the mathematical function name and the required accuracy, and outputs optimized C code. This generator enables users to customize the output precision based on the specific embedded application by using a static error analysis. This error analysis is comprised of the approximation error reported by the Maple command and the quantization error induced by the fixed datapath. This integer mathematical library can be integrated with any embedded software. The proposed approach enables: (1) automating the selection of approximation method, bitwidth and polynomial degree, (2) generating a customizable mathematical library from user-defined parameters to result evaluation, and (3) optimizing the fixed-point software implementations for elementary function evaluation. We test the code generated by our framework by running on an embedded PowerPC processor.

Automation

Our Matlab based code generator enables automation from user-defined error requirement to any particular elementary functions. For a given error requirement, the generator first generates coefficients using the Maple symbolic library from Matlab for both polynomial approximation and rational approximation. In the next sub-section, we present the detail of the approximations and the operations for enabling custom precision. We provide both 32-bit and 64-bit internal integer computations in order to create the custom-precision library. In the approximations, we can get better performance by using fewer polynomial degrees, however we also get a higher approximation error. When the embedded processor evaluates a given function using the generated C code, the input data x is originally in floating-point format. The range reduction transforms this number into its integer representation by scaling. As shown in Figure 3, since the exponent field of IEEE single precision is biased 127, we can directly use the mantissa field (fraction part) of the input data. The details of the integer and floating-point conversion are shown in Figure 6. We provide an analytical error analysis and define the error terms. For example, the approximation error is introduced by the Maple command when we generate the polynomial coefficients, and the quan-

C-code: Data structure for input/output
```
typedef struct sng_flds {
unsigned sgn : 1; // 0x8000 0000
unsigned exp : 8; // 0x7F80 0000(bias 127)
unsigned val : 23; // 0x007F FFFF
} SNG_FLD;
```

C-code: $fp2int$ - floating-point to integer
```
output = input.val << 8;
output += 0x80000000;
output = output >> 1;
```

C-code: $int2fp$ - integer to floating-point
```
input = (input >= 0)?
(sign = 0, input) : (sign = 1, -input);
exp = check_leading_zero(input);
input = input << exp;
output.val = (input & 0x80000000) >> 8;
output.exp = 129 - exp;
output.sgn = sign;
```

Figure 6: $fp2int$ and $int2fp$ operations (floating point and integer data conversions).

tization error is introduced in the datapath since infinite precision is required to represent the exact value.

Generation

IMGen uses the calculated maximum errors including approximation and quantization errors to determine the best possible library for a given error requirement. The output of the library construction includes: a library file, a reported generation time, an error upper bound, cycle counts from the hardware timer, an actual measured speedup from the embedded system when comparing with the Xilinx emulated math library and code space usage of the embedded C library. Lee et al. [15] show how we can achieve elementary function approximation using hardware, in this work we can apply the same principle to include more elementary functions. An important step in generating different functions is efficient code generation for range reduction and range reconstruction.

Optimization

In the code generator, several optimization techniques are used such as loop unrolling. By unrolling loops, we simplify decimal point correction. The basic idea is to make use of the information collected during Matlab code generation. In other words, the integer computations inside the embedded processor are pre-computed, as a result we achieve a performance gain. In Figure 3, we describe two conventional ways of implementing two functions evaluated in this paper. We show how we achieve range reduction, function approximation and range reconstruction. As shown in the figure, we use a degree-3 polynomial approximation when $f(x) = \log(x)$ and we use a degree-2 rational approximation when $f(x) = \sqrt{x}$.

The input x is first stored in a union data structure which has both $floating-point$ (fp) and $bit-selective$ elements such that we can easily select the sign-bit, mantissa and exponent of the input data. The IEEE single and double representations are used in the data structures. Next, we correct the exponent value since it is biased 127, and transform the input x into the corresponding integer x (which is y) in the figure. Finally, for the range reconstruction, different techniques are applied according to the input function.

Table 2: Main types of embedded PowerPC arithmetic and logical instructions.

Move	mr rD, rA
Add Carrying	addc rD, rA, rB
Add Extended	adde rD, rA, rB
Count Leading Zeros Word	cntlzw rA, rS
Multiply Low Word	mullw rD, rA, rB
Multiply High Word	mulhw rD, rA, rB
OR immediate	ori rA, rS, UIMM
Shift Left Word	slw rA, rS, rB
Shift Right Algebraic Word	sraw rA, rS, rB
Shift Right Word	srw rA, rS, rB
Subtract from Immediate Carrying	subfic rD, rA, SIMM

4.2 Polynomial Approximation

The generated floating-point coefficients are stored in an array and are scaled up to integers which maximizes the use of the 32-bit word size. Horner's rule is used to evaluate the polynomials. An adaptive datapath bit-width optimization is used to adjust the datapath for a correct decimal point in order to avoid any data overflow or underflow. As shown in Figure 4, the input floating-point x is transformed into its equivalent integer format by $fp2int()$, and the final integer result is converted back to the floating-point domain by $int2fp()$.

4.3 Rational Approximation

Rational approximation can be represented as follows:

$$f(x) = \frac{((c_n x + c_{n-1})x + \dots c_1)x + c_0)}{((d_n x + d_{n-1})x + \dots d_1)x + d_0)}$$

In the code generation, the degrees of the numerator and the denominator are the same. The major bottleneck is the final division at the end of the evaluation. A small degree of rational approximation can always achieve a similar or better accuracy than a relatively higher degree in polynomial approximation. Figure 5 shows a second-degree rational approximation with two multiplications, two additions and one division.

4.4 Custom Precision Code

We use fixed-point (integer) arithmetic for all internal calculations. First, we obtain the fractional bits from the input value and scale it up to the corresponding integer value, while an adaptive scaling method for the polynomial coefficients is used to avoid the mis-calculation. The important point is to maintain the correct "virtual" decimal point. In the following sub-sections, the input/output conversions and the internal basic operations are described. Note that we only need to modify these integer arithmetic operations for different integer processors. In Figure 6, the 32-bit floating-point to integer conversions are described. For example, the input data is corrected by adding the hidden MSB bit. The 64-bit conversion is not shown here but it basically uses the same principle.

PowerPC assembly: $s4.s5 = s2.s3 \ll s1$

```
msl(s1, s2, s3, s4, s5);
r21 = s1, r22 = s2, r23 = s3
subfic r11, r21, 32;
slw r22, r22, r21;
srw r20, r23, r11;
or r22, r22, r20;
addi r11, r21, -32;
slw r20, r23, r11;
or r22, r22, r20;
slw r23, r23, r21;
s4 = r22, s5 = r23
```

Figure 7: 64-bit integer shift left operation using 32-bit registers (s1,s2,s3,s4,s5).

PowerPC assembly: $s4.s5 = s2.s3 \gg s1$

```
msar(s1, s2, s3, s4, s5);
r21 = s1, r22 = s2, r23 = s3
subfic r11, r21, 32;
srw r23, r23, r21;
slw r20, r22, r11;
or r23, r23, r20;
addic. r11, r21, -32;
sraw r20, r22, r11;
ble $+8;
ori r23, r20, r20;
sraw r22, r22, r21;
s4 = r22, s5 = r23
```

Figure 8: 64-bit shift algebraic right operation using 32-bit registers (s1,s2,s3,s4,s5).

Add Operation

In Table 2, the main instructions that are used for **IMGen** are shown. The supported integer instructions include arithmetic, logical, compare, rotate and shift instructions. For example, the add carrying instruction *addc* adds the contents of register rA and rB to register rD. The latency of most instructions is one cycle, while the latencies for multiply and divide are 4 cycles and 35 cycles, respectively. Since PowerPC405 is an integer processor, system software requires floating-point emulation which is a call interface to subroutines within a floating-point run-time library. The subroutines emulate the operation of floating-point instructions.

Addition and multiplication are the two major operations for function evaluation. In **IMGen**, we support both 32-bit and 64-bit implementations. Note we add two 64-bit data where $s3 = s1 + s2$. Also, PowerPC405 is a 32-bit processor and the general purpose registers are all 32-bits, we need to use the add carrying and the add extended instructions to complete this operation. The carry from *addc* is added to the consecutive *adde* operation. Note that the same idea can be generalized to any 8-bit or 16-bit processor and to custom output precisions.

Multiply Operation

In the code generator, we provide two library functions that are used for 32-bit and 64-bit integer multiplication. If the design has a lower accuracy requirement, a 32-bit implementation and the "multiply-high-word" $mhw()$ operation is used. On the other hand, if a 64-bit datapath is selected, $s1$, $s2$ and $s3$ are stored into the corresponding MSB and LSB registers and the "multiply-low-word" $mlw()$ operation is also used. For example, the two input values are moved to two general purpose registers, $r21$ and $r22$, and then perform the embedded assembly instructions "mullw" and "mulhw", the final result is placed at the $r23$ register and thus finally move back to the $s3$ variable.

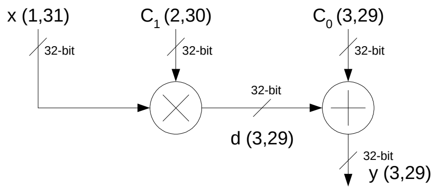


Figure 9: Degree one approximation for $\log(x)$.

Divide Operation

An optimized division operation is crucial for the rational approximation. We cannot directly use the 32-bit divide instruction from PowerPC since the result of the integer division for our case is always zero. Moreover, this division instruction uses a sequential divider in hardware and is very slow when compared to other instructions. As a result, we adopt the "shift-and-subtract" method [21] to perform this division. Our division simply uses "add", "shift" and "negate" instructions.

Shift Operations

Our adaptive datapath method corrects the decimal point involved in the polynomial and rational approximations. Depending on the input coefficients, the input range (the integer part of the coefficient) can be very large and we need to scale up the coefficient with a smaller scaling factor. This scaling operation is performed by bit shifting. In the proposed library, we have provided six shift operations: shift left, shift right and shift algebraic right for both 32-bit and 64-bit designs. As shown in Figure 7 and Figure 8, the 64-bit shift left and shift algebraic right are provided. For example, the $s2.s3$ refers to the MSB of the 64-bit data storing at $s2$ while the LSB storing in $s3$. The final result is shifted by $s1$ bits and is stored into $s4.s5$.

5. ERROR ANALYSIS

We divide the error analysis into two parts: approximation error and quantization error which is further divided into three parts: range reduction, function approximation and range reconstruction.

Approximation Error

Error is introduced in the function approximation step because approximation error is the error of approximating a function using polynomials that is independent of the precision we used. First, the Maple command we used to generate the polynomial coefficients reports this approximation error. Second, the quantization in the datapath since infinite pre-

Table 3: Approximation errors using polynomial and rational methods reported by Matlab for $\log(x)$.

Degree	Polynomial approx.	Rational approx.
1	0.02983005	0.0008607941
2	0.00342398	0.0000017146
3	0.00044161	0.0000000032

cision is insufficient to represent the exact value and the datapath in our designs is either 32-bit or 64-bit. We aim to maintain the highest accuracy with the available bitwidth and try to minimize the accumulated quantization error.

$$E_{total} = E_{approximation} + E_{quantization}$$
$$E_{quantization} = E_{poly\_approximation} + E_{range\_reconstruction}$$

The above two equations summarize the key ideas of this error analysis. The quantization error has three parts: range reduction, function approximation and range reconstruction. The range reduction step of the evaluated functions is exact where no quantization is involved as shown in Figure 3. We summarize the approximation errors reported by Matlab in Table 3. We can see that the approximation error generated by using the rational method is much lower than the polynomial method. Note that for example, degree-3 means that the degree of both the numerator and the denominator are three. In this error analysis, we only consider the polynomial approximation step and the range reconstruction step.

Quantization Error

Much research work focuses on the bit-width optimization problem, especially for the fraction bit-width optimization on hardware design [15]. These approaches mainly minimize the hardware bit-width in order to meet the error requirement due to the area and latency constraints. In this paper, the problem we are solving is constrained by the fixed datapath. As shown in Figure 9, since all the datapaths in this calculation are fixed to 32-bits, the proposed adaptive datapath method selects the best position of the decimal point. For example, we use one bit to store the integer part and 31 bits to store the fraction part of x. The Maple command which generates polynomial coefficient C_1 needs two bits for the integer part, as a result the multiplication product d needs three bits for the integer part. When we perform an addition, the two operands need to have the same decimal point, thus the final fraction bits for the $y = f(x)$ equals 29 bits. As shown below, we describe how we perform this static error analysis in the library generation.

We denote E_{signal} as the error of a particular signal. The polynomial approximation error bound of the signals in Figure 9 is described in Figure 10. The error bound induced by the reconstruction is given in Figure 11. The floating point value, input x is exactly transformed to its integer format, thus there is no quantization error in the input. Note that the static error analysis includes the quantization error in the computation, but not the approximation error by the Maple approximation. There are two main ways to quantize a signal: truncation and round-to-nearest. Here we define FB as the fractional bitwidth. Truncation and round-to-nearest can cause a maximum error of 2^{-FB} and 2^{-FB-1}. As shown in the above equations, the coefficients C_0 and

$$\boxed{\begin{aligned}
&E_y = E_{poly\_approximation}\\
&E_y = E_{accum} + E_{quantization_y}\\
&E_y = (E_d + E_{C_0}) + 2^{-29}\\[1ex]
&E_d = E_{C_1} + 2^{-29}\\
&E_{C_1} = 2^{-31}\\
&E_{C_0} = 2^{-30}\\[1ex]
&Therefore,\\
&E_y = 2^{-31} + 2^{-29} + 2^{-30} + 2^{-29}\\
&E_y = 5.12227 \times 10^{-9}
\end{aligned}}$$

Figure 10: Static error analysis of degree-one $\log(x)$.

$$\boxed{\begin{aligned}
&E_{range\_reconstruction} = E_{\log 2} \times exp\\
&Overflow_{max} = 5\\[1ex]
&E_{\log 2} = 2^{-31}\\
&E_{overflow} = E_{\log 2} \times 2\\
&E_{accum} = E_{\log 2} \times E_{overflow}\\
&E_{accum} = 2^{-31} + 2^{-30} + 2^{-29} + 2^{-28} + 2^{-27}\\[1ex]
&Therefore,\\
&E_{range\_reconstruction} = 4.1607 \times 10^{-9}
\end{aligned}}$$

Figure 11: Error analysis of $\log(x)$ range reconstruction.

C_1 are rounded to the nearest, and the error at the output $y = f(x)$ is the total of accumulated error and the quantization error of this signal. The second part of the quantization error belongs to the range reconstruction.

Below, we show the minimum error bounds provided by the IEEE single and double precision formats. It means that if the final computed error is smaller than these two values, the library/design can assure the accuracy requirement of either IEEE single or double precision.

$$E_{single} = 2^{-23} = 1.192 \times 10^{-7}$$
$$E_{double} = 2^{-52} = 2.220 \times 10^{-16}$$

Now that the quantization error of the above example is lower than E_{single}, however the approximation error of this example is $E_{approx} = 0.02983$ as shown in Table 3 which is much higher than E_{single}. We apply this static analysis in our code generation to cover all the possible errors in the calculation for determining the best possible embedded C code.

6. OPTIMIZATION

This section discusses the optimization techniques we use for code generation, and also for code space optimization.

Optimized Code Generation

One of the code generation optimizations we use is loop unrolling. As shown in Figure 12, two code fragments of degree-one function evaluation are used to illustrate the basic idea. the array *ic* is used to hold the polynomial coefficients. `CORRECTION` represents the adaptive datapath correction for adjusting the decimal point. Figure 17 shows that when we use a higher polynomial degree, a smaller approxi-

```
    long ic[2] = {1488522235, -1456492463};
    #define CORRECTION if (j==0) s3 = s3 << 1;

  unoptimized code
    ix = fp2int(input);
    iy = ic[0];
    for (j=0; j<degree; j++){
       mhw(ix, iy, s3);
       CORRECTION
       iy = s3 + ic[j+1];
    }

  using loop-unrolling technique
    ix = fp2int(input);
    mhw(ix, ic[0], s3);
    s3 = s3 << 1;
    iy = s3 + ic[1];
```

Figure 12: Code generation optimization example.

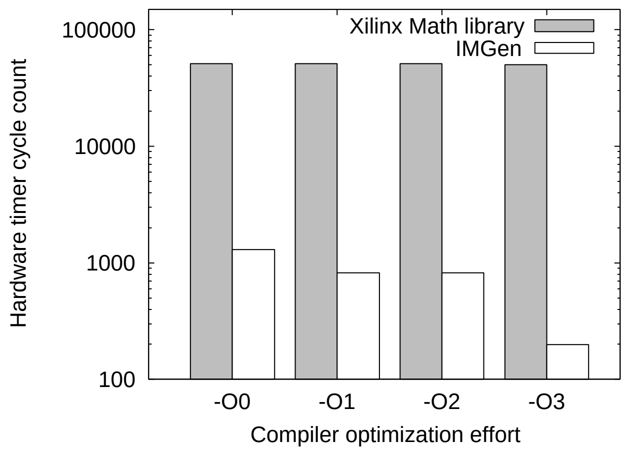


Figure 13: The effect of compiler optimization on library performance for $\log(x)$ function evaluation.

mation error is obtained while the accumulated quantization error is increased. This U-shaped curve not only describes the effect of changing the polynomial degree, but also shows the effect of using code generation optimization. The maximum error shown in the graph represents maximum error of the empirical study of thousands of sample data against the Matlab result. These code generation optimization techniques are also applied to 64-bit datapath designs and rational approximation designs.

Compile Time Optimization

Besides the code generation optimization, compilation techniques such as code reuse and register renaming are extremely important for instruction processors. Figure 13 shows that the existing emulated math library is highly optimized since further compilation optimization does not affect the overall performance. On the other hand, the generated **IMGen** library is compiled with the highest optimization in order to compare with the emulated math library.

```
User input >> genlib('log', 0.01)

Phase 1: Maple command generates polynomial coefficients
Phase 2: Static error analysis calculates quantization error
Phase 3: Select polynomial/rational approximation
Phase 4: Select 32-bit/64-bit implementation
Phase 5: Generate embedded C code
         and execute in embedded integer processor
Phase 6: Output performance data and statistical error
       text  data  bss    dec   hex  filename
      44232  4296   48  48576  bdc0  TestApp/executable.elf
      cycle count for the Xilinx math library: 63335
      cycle count for the bus overhead:        60
      cycle count for the IMGen library:       618
      average speedup: 1.13e+002
      maximum error  : 0.0034241
      IMGen is generated and tested in 4.904e+001 seconds
```

Figure 14: Input/Output of the automated system.

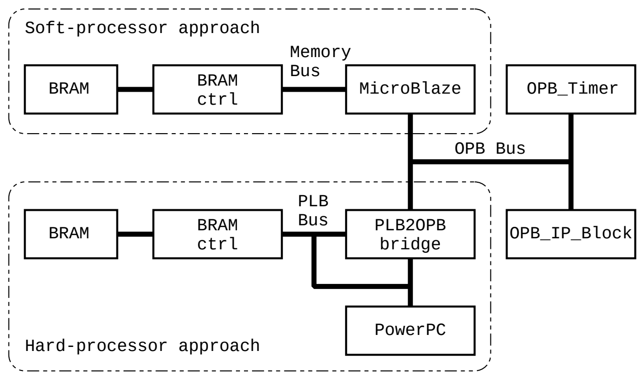


Figure 15: Our embedded system under testing. (PLB: processor local bus, OPB: on-chip peripheral bus, and BRAM: block RAM.)

Polynomial Degree Automation

The inputs of the IMGen generator are the mathematical function name and the error requirement. As shown in Figure 14, the Matlab interface passes the parameters for the error analysis as described in the previous section. The embedded C code is generated and evaluated in the prototyping system with the best execution time and accuracy tradeoff. In the same figure, it shows that the Matlab function to generate the library is for example "**genlib('log',0.01)**". The polynomial degree and the design methods are automatically selected by the library generator based on the input function and also the user-defined error requirement.

Instruction Code Space Optimization

The change of the polynomial degree is independent of the code space usage for the emulated math library, whereas the code space of **IMGen** increases gradually with a higher polynomial degree. In this work, we demonstrate that the proposed framework not only provides a flexible and efficient custom-precision math library, but also reduces the required instruction code space which is very expensive in embedded systems. The major reason for the reduction is the fact that our method avoids the floating point to integer conversions.

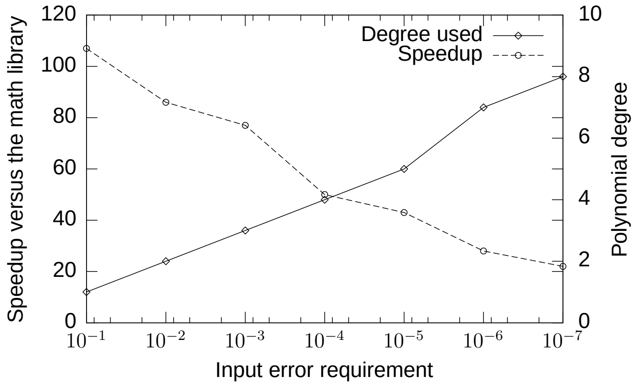


Figure 16: 32-bit $\log(x)$ function evaluation without optimization. Speedup comparison between the Xilinx emulated floating point math library (precision: IEEE single) and IMGen (precision: variable).

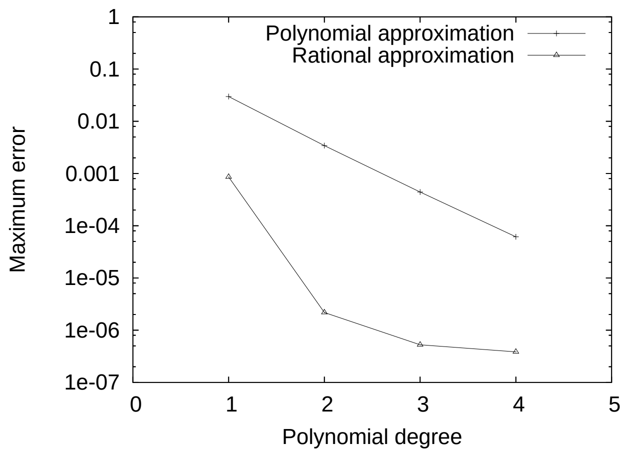


Figure 18: Maximum error comparison using polynomial method and rational method of $\log(x)$.

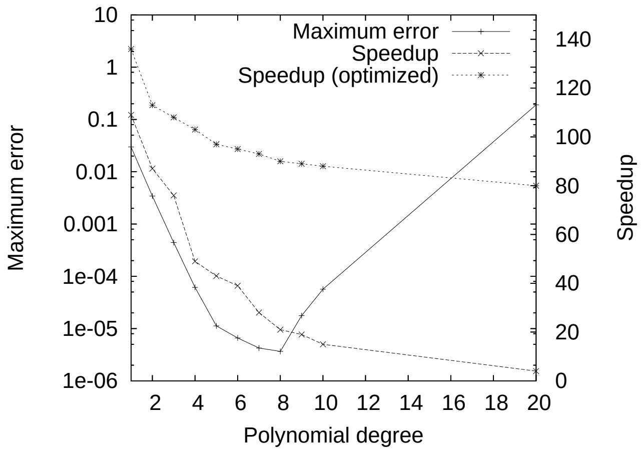


Figure 17: U-shaped curve for the 32-bit approximation and quantization errors of $\log(x)$.

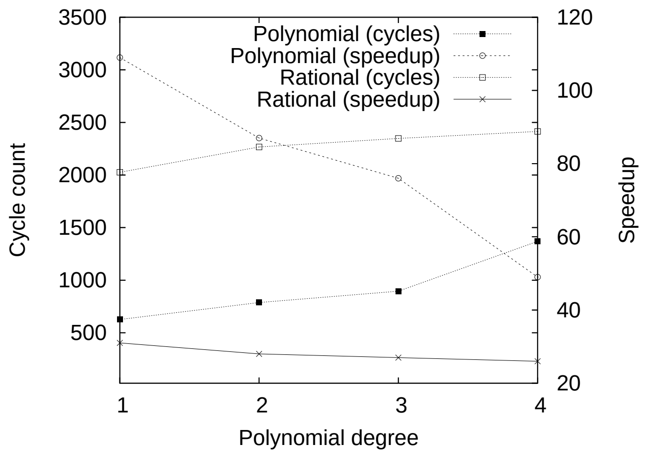


Figure 19: Speedup comparison using polynomial method and rational method of $\log(x)$.

7. RESULTS

This section presents the experimental setup and evaluation of the proposed approach using different methods.

Experimental Setup

We evaluate the proposed design flow using the Xilinx ML310 system which has an XC2VP30 Virtex-II Pro device [4] embedded with two PowerPC processors. The embedded system also enables prototyping of the soft-processor approach as shown in Figure 15. The OPB_Timer is connected to the embedded PowerPC through the OPB and PLB buses. The measured cycle counts refer to the clock cycles of the OPB_Timer for the function evaluation.

Evaluation

As shown in Figure 16, the result of evaluating $\log(x)$ in an embedded processor shows that over 100 times speedup is achieved when the error requirement is higher than 10^{-3}. In this paper, we use the OPB timer to measure the exact time for computation. The embedded processor sends the start and done signals to the timer before and after function evaluation. The register in the timer records the exact number of clock cycles. In order to measure bus latency, we send a `start` signal followed by a `stop` signal to the timer without running any extra instructions in the embedded processor. This OPB bus latency is defined as $T_{overhead}$, $T_{math\_library}$ is the time taken to evaluate functions in the emulated math library, and T_{IMGen} is the time (in cycle counts) spent in **IMGen**.

The equation below describes how we measure the speedup using the OPB_Timer. We also randomly generate a large testbench dataset in order to statistically measure the speedup. Looking at Figure 17, we can see that significant speedup is achieved when optimization techniques are applied to code generation. Figure 18 shows that the rational method achieves much lower error than the polynomial method for small degree. Note that the maximum error values shown in this graph are much higher than those in Table 3. This is be-

Table 4: Comparisons of 32-bit, 32-bit-opt and 64-bit implementations using degree-4 polynomial method.

	32-bit	32-bit (opt.)	64-bit
Xilinx Math (cycles)	63759	63699	64370
Bus latency (cycles)	60	60	60
IMGen (cycles)	1369	672	1921
Speedup factor	48x	103x	34x
Measured error	0.00005886	0.00005920	0.00000159

Table 5: Comparisons of $\log(x)$ and $\sqrt{x}$ polynomial approximations using degree-6 polynomial method.

	$\log(x)$	$\sqrt{x}$
Xilinx Math (cycles)	62725	9159
Bus latency (cycles)	60	60
IMGen (cycles)	1696	467
Speedup factor	38x	22x
Measured error	0.000004313	0.0008375

Table 6: Comparisons of $\log(x)$ and $\sqrt{x}$ polynomial approximations with reference designs [12].

	Intel XScale^{TM} [12]		IMGen
	Single precision	Double precision	10^{-5}
	Latency [cycles]	Latency [cycles]	Timer [cycles]
$\sqrt{x}$	84	183	587
$\log(x)$	215	433	710

cause the dominating factor of quantization error varies with the degree value. On the other hand, in Figure 19 we observe that the penalty of the division used in the rational method is high when compared to the polynomial method. The proposed design flow satisfies custom precision requirements for most space expensive embedded systems. The division in the rational method also limits the speedup factor.

$$Speedup = (T_{math\_library} - T_{overhead})/(T_{IMGen} - T_{overhead})$$

Given that we have a customized precision math library, what is the overall speedup in real world applications? To answer this question, we first need to profile the embedded system program. Then according to Amdahl's law, overall speedup $= \frac{1}{1-P+P/S}$ where P is the improved computation portion, and S is the speedup factor for that improved portion of the system. Now, suppose that over 60% of the total run time is spent on function evaluations in a graphics/multimedia application, and the speedup factor is 100. The system overall speedup is 2.463.

It is obvious that using a 64-bit datapath can achieve a higher precision, however the more internal computations the slower the overall function evaluation. Here, we compare the accuracy and performance using 32-bit and 64-bit datapaths. Table 4 shows some results using the degree-4 polynomial method. We can see that, (1) the number of clock cycle measured by the OPB_Timer for both Xilinx math library and using our code generator, (2) and the 64-bit measured error is much lower than the 32-bit one. In the table, "Bus latency" refers to $T_{overhead}$ which is the OPB bus latency. The execution time of each elementary math function routine varies depending on the function nature and the input arguments. For example, as shown in Figure 5 the execution time of $\log(x)$ is almost 6 times slower than $\sqrt{x}$. Although we use the degree-6 polynomial method for both evaluations, the speedup factor and the measured error are both highly dependent on the input function and input arguments. We also compare our designs with the Intel XScale^{TM} math library. Looking at Table 6, the latencies in terms of the XScale processor clock and the OPB_Timer are described. Our current designs are slower than the highly optimized Intel library, mainly because of its specialized range reduction techniques and an efficient table-lookup and software implementation. However, our approach demonstrates the flexibility of automatically generating customizable embedded C code for integer processors with different accuracy requirements.

8. CONCLUSIONS

This paper introduced a customizable library for floating-point function evaluation based on the integer instruction set used in embedded processors. We have developed an approach for automating code generation and optimization within this framework to customize precision in order to obtain the best trade-off in embedded code space, performance and accuracy. We evaluate this approach using the embedded PowerPC processor on a Xilinx XC2VP30 FPGA. By trading off accuracy, we achieve over 80 times speedup compared to the single-precision reference mathematical library with a static error analysis below 10^{-5}, while a 64-bit polynomial method achieves 30 times speedup when compared to IEEE single-precision. One of the current limitations of the framework is that the quantization error of the 64-bit method is higher than IEEE double-precision, we can use other function evaluation techniques such as piecewise polynomial approximation and multiple words in the datapath to achieve a higher accuracy.

Ongoing and future work includes code generation for other integer instruction processors and comparison with floating-point coprocessors such as the Xilinx floating-point coprocessor using the new Auxiliary Processor Unit that integrates hardware accelerators and co-processors with the PowerPC processor. Second, we plan to use better range reduction techniques targeting software implementations [12, 19]. We can also reconfigure a soft processor using FPGA technology at run-time to achieve the best tradeoff in speed, area and precision for specific applications.

9. ACKNOWLEDGEMENTS

The support of the Jet Propulsion Laboratory, Celoxica, Xilinx Inc., the Croucher Foundation and UK EPSRC (Grant number GR/R 31409, GR/R 55931, GR/N 66599) is gratefully acknowledged. The authors thank the assistance and comments of Andreas Fidjeland, Tero Rissa, Timothy Todman and Peter Collingbourne.

10. ADDITIONAL AUTHORS

Wayne Luk, (Department of Computing, Imperial College London, United Kingdom, email: w.luk@imperial.ac.uk) and Peter Y.K. Cheung, (Department of Electrical & Electronic Engineering, Imperial College London, United Kingdom, email: p.cheung@imperial.ac.uk).

11. REFERENCES

[1] *Accelerated System Performance with APU-Enhanced Processing, Xcell Journal, Xilinx Inc.* *http://www.xilinx.com/publications/xcellonline/xcell_52/xc_pdf/xc_v4acu52.pdf.*

[2] *Excalibur Device Overview Data Sheet, Altera Inc.* *http://www.altera.com/literature/ds/ds_arm.pdf.*

[3] *MicroBlaze Processor Reference Guide, Xilinx Inc.* *http://www.xilinx.com/ise/embedded/mb_ref_guide.pdf.*

[4] *ML310 User Guide, Xilinx Inc.* *http://www.xilinx.com/products/boards/ml310/current/pcb/sch/ug068.pdf.*

[5] *PowerPC 405 ProcessorBlock Reference Guide, Xilinx Inc. http://www.xilinx.com/bvdocs/userguides/ppc405block_ref_guide.pdf.*

[6] A. Akkaş and M.J. Schulte. "A quadruple precision and dual double precision floating-point multipler". In *IEEE Symp. on Digital System Design*, pages 76–81, 2003.

[7] R. Brent. "Fast multiple-precision evaluation of elementary functions". *Journal of the Association for Computing Machinery*, 23(2):242–251, 1976.

[8] N. Brisebarre, D. Defour, P. Kornerup, J. Muller, and N. Revol. "A new range-reduction algorithm". *IEEE Trans. on Computers*, 54(3):331–339, 2005.

[9] M. Ercegovac, T. Tang, J. Muller, and A. Tisserand. "Reciprocation, square root, inverse square root, and some elementary functions using small multipliers". *IEEE Trans. on Computers*, 49(7):628–637, 2000.

[10] S. Gal and B. Bachelis. "An accurate elementary mathmetical library for the IEEE floating point standard". *ACM Trans. on Mathematical Software*, 17(1):26–45, 1991.

[11] Y. Hida, X. Li, and D. Bailey. "Algorithms for quad-double precision floating point arithmetic". In *Proc. IEEE Symp. on Computer Arithmetic*, pages 155–162, 2001.

[12] C. Iordache and P. Tang. "An overview of floating-point support and math library on the Intel XScale^{TM} architecture". In *Proc. IEEE Symp. on Computer Arithmetic*, pages 122–128, 2003.

[13] J. Harrison and T. Kubaska and S. Story and P.T.P. Tang. "The computation of transendental functions on the IA-64 architecture". *Intel Technology Journal, Q4*, pages 1–7, 1999.

[14] I. Koren and O. Zinaty. "Evaluation of elementary functions in a numerical co-processor based on rational approximations". *IEEE Trans. on Computers*, 39:1030–1037, 1990.

[15] D. Lee, A. Abdul Gaffar, O. Mencer, and W. Luk. "Adaptive range reduction for hardware function evaluation". In *Proc. IEEE Int'l Conf. on Field-Programmable Technology*, pages 169–176, 2004.

[16] T. Lynch, A. Ahmed, and M. Schulte. "The K5 transcendental functions". In *Proc. IEEE Symp. on Computer Arithmetic*, pages 163–170, 1995.

[17] P. Markstein. "Computation of elementary functions on the IBM RISC System/6000 processor". *IBM Journal Res. Develop.*, 34(1):111–119, 1990.

[18] O. Mencer and W. Luk. "Parameterized high throughput function evaluation for FPGAs". *Journal of VLSI Sig. Proc. Syst.*, 36(1):17–25, 2004.

[19] J. Muller. *"Elementary functions: algorithms and implementation"*. Birkhauser Verlag AG, 1997.

[20] A. Nesterov. *"Optimized math library for TMS320C67x DSP reference manual"*, 2001.

[21] B. Parhami. *"Computer Arithmetic: Algorithms and Hardware Designs"*. Oxford University Press, 2000.

[22] G. Paul and W. Wilson. "Should the elementary function library be incorporated into computer instruction sets". *ACM Trans. on Mathematical Software*, 2(2):132–142, 1976.

[23] M. Schulte and E.E. Swartzlander Jr. "Hardware designs for exactly rounded elementary functions". *IEEE Trans. on Computers*, 43(8):964–973, 1994.