Automating Custom-Precision Function Evaluation for Embedded Processors

Ray C.C. Cheung, Dong-U Lee*, Oskar Mencer, Wayne Luk and Peter Y.K. Cheung**

Department of Computing, Imperial College
Electrical Engineering Department, UCLA*
Department of EEE, Imperial College**

CASES
September 2005
Talk outline

1. achievements
2. motivation
3. function evaluations
4. design tool flow
5. error analysis
6. performance evaluation
7. future work
8. summary
1. Achievements

• customizable library for floating-point function evaluation based on input integer instruction set

• automatic code generation using high-level Matlab model, and optimization for customizing precisions

• evaluation of this method with two elementary functions and Xilinx embedded design kit

• automating the selection of approximation method, polynomial degree for a given function, accuracy requirement and execution time
2. Motivation

- embedded systems are usually space and time critical, a dedicated co-processor and a larger memory for instruction are infeasible
- previous work on math co-processor and floating point emulation
- automated code generation for mathematical function library targeting customizable precision (depending on the error requirements)
3. Function approximation

- Polynomial method: approximation with a single polynomial
Rational approximation

- Rational method: with two polynomials (same degree)
Range Reduction

- \( f(x) \) where \( x = [a, b] \)
  - (1) range reducing \( x \) to a more convenient interval \( y = [a', b'] \)
  - (2) function approximation on the reduced interval
  - (3) range reconstruction: expanding the result back to the original result range
Example: Evaluating log(x)

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 = ((c3 × y + c2) × y + c1) × y + c0;

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

input x

Adjust the output exponent
4. Design tool flow using Matlab

- technology-independent flow
  – use the embedded PowerPC as an example

- Integer processor instruction set
- Select approximation method and polynomial degree
- Code optimization and adaptive datapath correction
- Generate code with the best degree and performance
- Compare error and speed improvement

Function f(x) Error requirement

C code

Embedded integer processor
IMGen – 3 steps

• automation:
  – user error requirement → function evaluation implementation
  – select rational / polynomial approximation
  – select rational / polynomial degree
  – select 32-bit / 64-bit datapath

• generation: using custom precision code
  – Add / multiply / divide / shift operations

• optimization: e.g. loop unrolling techniques
Code optimization

• code generation optimization example

```c
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];
```
Floating-point – fixed-point conversion

- input and output are both floating-point format
- internal computation is transparent to users

**C-code:** Data structure for input/output
```c
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
```c
output = input.val << 8;
output += 0x80000000;
output = output >> 1;
```
5. Error analysis

- approximation error (error of approximating a function, e.g. using minimax)
- quantization error induced by: (1) the multiply-add datapath, (2) range reconstruction (function dependent)
- rational approximation has a much lower approximation error

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

<table>
<thead>
<tr>
<th>Degree</th>
<th>Polynomial approx.</th>
<th>Rational approx.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.02983005</td>
<td>0.00008607941</td>
</tr>
<tr>
<td>2</td>
<td>0.00342398</td>
<td>0.00000017146</td>
</tr>
<tr>
<td>3</td>
<td>0.00044161</td>
<td>0.00000000032</td>
</tr>
</tbody>
</table>
Error analysis example

- quantization error analysis of degree-one log(x)
- $x(1,31) = 31$ fraction bits
- $E_d = \text{error accumulated at signal } d$
- higher degree $\rightarrow$ higher error

$$
E_y = E_{\text{poly approximation}}
E_y = E_{\text{accum}} + E_{\text{quantization}_y}
E_y = (E_d + E_{C_0}) + 2^{-29}

E_d = E_{C_1} + 2^{-29}
E_{C_1} = 2^{-31}
E_{C_0} = 2^{-30}

\text{Therefore,}
E_y = 2^{-31} + 2^{-29} + 2^{-30} + 2^{-29}
E_y = 5.12227 \times 10^{-9}
$$
System automation

• input / output via Matlab, remote execution on the embedded system board

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
Embedded system under test

- use Xilinx ML310 system, XC2VP30 device, with two embedded PowerPC chips
- can target Xilinx MicroBlaze soft integer processor
6. Performance evaluation

- compare with Xilinx emulated math library
Compile time optimization

- study the effect of compiler optimization

![Bar chart showing compile-time optimization effects]
Polynomial vs. rational

- we measure the bus latency, measure an accurate speedup factor

\[ \text{Speedup} = \frac{T_{\text{math library}} - T_{\text{overhead}}}{T_{\text{IMGen}} - T_{\text{overhead}}} \]
Performance comparisons

<table>
<thead>
<tr>
<th></th>
<th>32-bit</th>
<th>32-bit (opt.)</th>
<th>64-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Math (cycles)</td>
<td>63759</td>
<td>63699</td>
<td>64370</td>
</tr>
<tr>
<td>Bus latency (cycles)</td>
<td>60</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>IMGGen (cycles)</td>
<td>1369</td>
<td>672</td>
<td>1921</td>
</tr>
<tr>
<td>Speedup factor</td>
<td>48x</td>
<td>103x</td>
<td>34x</td>
</tr>
<tr>
<td>Measured error</td>
<td>0.00005886</td>
<td>0.00005920</td>
<td>0.00000159</td>
</tr>
</tbody>
</table>

tradeoff between speed and accuracy

<table>
<thead>
<tr>
<th></th>
<th>log (x)</th>
<th>(\sqrt{x})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx Math (cycles)</td>
<td>62725</td>
<td>9159</td>
</tr>
<tr>
<td>Bus latency (cycles)</td>
<td>60</td>
<td>60</td>
</tr>
<tr>
<td>IMGGen (cycles)</td>
<td>1696</td>
<td>467</td>
</tr>
<tr>
<td>Speedup factor</td>
<td>38x</td>
<td>22x</td>
</tr>
<tr>
<td>Measured error</td>
<td>0.0000004313</td>
<td>0.0008375</td>
</tr>
</tbody>
</table>
7. Future work

- code generation for more integer processors
- comparison with floating-point coprocessor
- use better range reduction technique for software implementation
- use run-time reconfiguration to configure soft-processors such as MicroBlaze
8. Summary

- customizable library for floating-point function evaluation based on input integer instruction set
- automatic code generation using high-level Matlab model, and optimization for customizing precisions
- evaluation of this method with two elementary functions and Xilinx embedded design kit
- automating the selection of approximation method, polynomial degree for a given function, accuracy requirement and execution time
- embedded code generator
  - cope with speed/code-size/error trade-off