An SMT Based Method for Optimizing Arithmetic Computations in Embedded Software Code

Hassan Eldib and Chao Wang

FMCAD, October 22, 2013
The Dream

• Having a tool that automatically synthesizes the optimum version of a software program.
Embedded Software
Objective

- Synthesizing an optimal version of the C code with fixed-point linear arithmetic computation for embedded devices.
  - Minimizing the bit-width.
  - Maximizing the dynamic range.
Motivating Example

- Compute average of $A$ and $B$ on a microcontroller with signed 8-bit fixed-point

- Given: $A, B \in [-20, 80]$.

  - $\frac{A+B}{2}$ may have overflow errors.
  - $\frac{A}{2} + \frac{B}{2}$ may have truncation errors.
  - $B + \frac{A-B}{2}$ has neither overflow nor truncation errors.
Larger range requires a larger bit-width.
Decreasing the bit-width, will reduce the range.
Fixed-point Representation

Representations for 8-bit fixed-point numbers

- **Range:** -128 ↔ 127
- **Resolution:** 1

- **Range:** -16 ↔ 15.875
- **Resolution:** 1/8

\[ \text{Range} \propto \text{Bit-width} \]

\[ \text{Resolution} \propto \text{Bit-width} \]
Problem Statement

Program:

```
1: int comp(int A, int B, int H, int E, int D, int F, int K) {
2:    int t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10, t11, t12;
3:    t12 = 3 * A;
4:    t10 = t12 + B;
5:    t11 = H << 2;
6:    t9 = t10 + t11;
7:    t6 = t9 >> 3;
8:    t8 = 3 * E;
9:    t7 = t8 + D;
10:   t5 = t7 - 16469;
11:   t3 = t5 + t6;
12:   t4 = 12 * F;
13:   t2 = t3 - t4;
14:   t1 = t2 >> 2;
15:   t0 = t1 + K;
16:   return t0;
17:}
```

Range & resolution of the input variables:

A -1000 3000
res. 1/4
B -1000 3000
res. 1/4
...

Optimized program:

```
1: int comp(int A, int B, int H, int E, int D, int F, int K) {
2:    int t0, t1, t3, t4, t5, t6, t8, t12;
3:    int N1, N2, N3, N4, N5, N6, N7, N9, N10;
4:    t12 = 3 * A;
5:    N6 = H;
6:    N10 = t12 - B;
7:    N9 = N10 >> 1;
8:    N7 = B + N9;
9:    N5 = N7 >> 1;
10:   N4 = N5 + N6;
11:   t6 = N4 >> 1;
12:   t8 = 3 * E;
13:   N3 = t8 - 16469;
14:   t5 = N3 + D;
15:   t3 = t5 + t6;
16:   t4 = 12 * F;
17:   N2 = t4 >> 2;
18:   N1 = t3 >> 2;
19:   t1 = N1 - N2;
20:   t0 = t1 + K;
21:   return t0;
22:}
```
Problem Statement

• Given
  – The C code with fixed-point linear arithmetic computation
  – The range and resolution of all input variables

• Synthesize the optimized C code with
  – Reduced bit-width with same input range, or
  – Larger input range with the same bit-width
SMT-based Inductive Program Synthesis

1. Program + Specs
2. Find a candidate program
3. Verify found program
4. Failed: Block program from appearing again
5. Passed: Synthesized program
Some Related Work

• Jha, 2011
  – Use an SMT solver to choose the best fixed-point representation in order to reduce error. No new programs are synthesized.

• Majumdar, Saha, and Zamani, 2012
  – Use a mixed integer linear programing (MILP) solver to minimize the error bound by only changing the fixed-point representation.

• Schkufza, Sharma, and Aiken, 2013
  – Use a compiler based method for optimization, which is an exhaustive approach.
SMT-based Inductive Program Synthesis

Program + Specs → Find a candidate program → Verify found program → Passed → Synthesized program

Failed → Block program from appearing again → Blocked programs
Step 1: Finding a Candidate Program

- Create **the most general AST** that can represent any arithmetic equation, with reduced **bit-width**.

- Use SMT solver to find a solution such that
  - For some test inputs (samples),
  - output of the AST is the same as the desired computation
SMT-based Solution

- SMT encoding for the general equation AST structure
  - Each $Op$ node can any operation from $*$, $+$, $-$, $>>$ or $<<$.
  - Each $L$ node can be an input variable or a constant value.
- SMT Solver finds a solution by equating the AST output to that of the desired program
SMT Encoding

- $\Psi = \Phi_{prog} \land \Phi_{AST} \land \Phi_{sameI} \land \Phi_{sameO} \land \Phi_{in} \land \Phi_{block}$
  
  - $\Phi_{prog}$: Desired input program to be optimized.
  - $\Phi_{AST}$: General AST with reduced bit-width.
  - $\Phi_{sameI}$: Same input values.
  - $\Phi_{sameO}$: Same output value.
  - $\Phi_{in}$: Test cases (inputs).
  - $\Phi_{block}$: Blocked solutions.
SMT-based Solution (an example)

\[
\frac{A}{2} + \frac{B}{2} \equiv \\
\begin{array}{c}
+ \\
>>
\end{array}
\begin{array}{c}
A \\
1 \\
B \\
1
\end{array}
\]
SMT-based Inductive Program Synthesis

Program + Specs

Find a candidate program

Verify found program

Passed

Synthesized program

Failed
Block program from appearing again

Blocked programs
Step 2: Verifying the Solution

• Is the program good for all possible inputs?
  – Yes, we found an optimized program
  – No, block this (bad) solution, and try again
SMT Encoding

- $\Phi = \Phi_{prog} \land \Phi_{sol} \land \Phi_{sameI} \land \Phi_{diffO} \land \Phi_{ranges} \land \Phi_{res}$
  - $\Phi_{prog}$: Desired input program to be optimized.
  - $\Phi_{sol}$: Found candidate solution.
  - $\Phi_{sameI}$: Same input values.
  - $\Phi_{diffO}$: Different output value.
  - $\Phi_{ranges}$: Ranges of the input variables.
  - $\Phi_{res}$: Resolution of the input variables.
SMT-based Inductive Program Synthesis

Program + Specs → Find a candidate program → Verify found program

Failed → Block program from appearing again

Passed → Synthesized program

Blocked programs
The Next Solution

\[ B + \frac{A-B}{2} \equiv \]

![Diagram showing a binary tree with operations involving A, B, and 1.]
SMT-based Inductive Program Synthesis
Scalability Problem

• Advantage of the SMT-based approach
  – Find optimal solution within an AST depth bound

• Disadvantage
  – Cannot scale up to larger programs

• Sketch tool by Solar-Lezama & Bodik (5 nodes)
• Our own tool based on YICES (9 nodes)
Incremental Optimization

• Combine static analysis and SMT-based inductive synthesis.

• Apply SMT solver only to small code regions
  – Identify an instruction that causes overflow/underflow.
  – Extract a small code region for optimization.
  – Compute redundant LSBs (allowable truncation error).
  – Optimize the code region.
  – Iterate until no more further optimization is possible.
Our Incremental Approach

Input program

For each node

If will flow

Yes

Extract region

Yes

Optimize

Successful?

Yes

Optimized program

No

Exit

Program + Specs

Find a candidate program

Verify found program

Passed

Blocked programs

Failed

Block program from appearing again

Synthesized program

Hassan Eldib and Chao Wang
The addition of $a$ and $b$ may overflow
Example

Computing Redundant LSBs

- The redundant LSBs of $a$ are computed as 4 bits
- The redundant LSBs of $b$ are computed as 3 bits.
Example

Extracting Code Region

- Extract the code surrounding the overflow operation.
- The new code requires a smaller bit-width.
Implementation

• Clang/LLVM + Yices SMT solver
• Bit-vector arithmetic theory
• Evaluated on a set of public benchmarks for embedded control and DSP applications
## Benchmarks (*embedded control software*)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Bits</th>
<th>LoC</th>
<th>Arithmetic Operations</th>
<th>Citation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sobel Image filter</td>
<td>32</td>
<td>42</td>
<td>28</td>
<td>Qureshi, 2005</td>
</tr>
<tr>
<td>Bicycle controller</td>
<td>32</td>
<td>37</td>
<td>27</td>
<td>Rupak, Saha &amp; Zamani, 2012</td>
</tr>
<tr>
<td>Locomotive controller</td>
<td>64</td>
<td>42</td>
<td>38</td>
<td>Martinez, Majumdar, Saha &amp; Tabuada, 2010</td>
</tr>
<tr>
<td>IDCT (N=8)</td>
<td>32</td>
<td>131</td>
<td>114</td>
<td>Kim, Kum, &amp; Sung, 1998</td>
</tr>
<tr>
<td>Controller impl.</td>
<td>32</td>
<td>21</td>
<td>8</td>
<td>Martinez, Majumdar, Saha &amp; Tabuada, 2010</td>
</tr>
<tr>
<td>Differ. image filter</td>
<td>32</td>
<td>131</td>
<td>77</td>
<td>Burger, &amp; Burge, 2008</td>
</tr>
<tr>
<td>FFT (N=8)</td>
<td>32</td>
<td>112</td>
<td>82</td>
<td>Xiong, Johnson, &amp; Padua, 2001</td>
</tr>
<tr>
<td>IFFT (N=8)</td>
<td>32</td>
<td>112</td>
<td>90</td>
<td>Xiong, Johnson, &amp; Padua, 2001</td>
</tr>
</tbody>
</table>

*All benchmark examples are public-domain examples*
Experiment *(increase in range)*

- Average increase in range is 307%
  
  (602%, 194%, 5%, 40%, 32%, 1515%, 0%, 103%)
Experiment \textit{(decrease in bit-width)}

<table>
<thead>
<tr>
<th>Name of Benchmark</th>
<th>Original (bit-width)</th>
<th>Optimized (bit-width)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Minimum</td>
<td>Average</td>
</tr>
<tr>
<td>Sobel image filter (3x3)</td>
<td>17</td>
<td>10.26</td>
</tr>
<tr>
<td>Bicycle controller</td>
<td>18</td>
<td>14.47</td>
</tr>
<tr>
<td>Locomotive controller</td>
<td>33</td>
<td>29.41</td>
</tr>
<tr>
<td>IDCT (N=8)</td>
<td>20</td>
<td>16.29</td>
</tr>
<tr>
<td>Control. Impl.</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>Diff. image filter (5x5)</td>
<td>17</td>
<td>11.11</td>
</tr>
<tr>
<td>FFT (N=8)</td>
<td>18</td>
<td>7.32</td>
</tr>
<tr>
<td>IFFT (N=8)</td>
<td>17</td>
<td>7.11</td>
</tr>
</tbody>
</table>

• Required bit-width: \textcolor{red}{32-bit} $\rightarrow$ \textcolor{red}{16-bit}  
\textcolor{red}{64-bit} $\rightarrow$ \textcolor{red}{32-bit}
Experiment \textit{(scaling error)}

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Scaling</th>
<th>Error original</th>
<th>Error optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sobel Image filter (3x3)</td>
<td>32-b $\rightarrow$ 16-b</td>
<td>$3.1 \times 10^{-2}$</td>
<td>0.0</td>
</tr>
<tr>
<td>Bicycle controller</td>
<td>32-b $\rightarrow$ 16-b</td>
<td>$3.5 \times 10^{-4}$</td>
<td>$2.0 \times 10^{-4}$</td>
</tr>
<tr>
<td>Locomotive controller</td>
<td>64-b $\rightarrow$ 32-b</td>
<td>$2.9 \times 10^{-8}$</td>
<td>$1.5 \times 10^{-9}$</td>
</tr>
<tr>
<td>IDCT (N=8)</td>
<td>32-b $\rightarrow$ 16-b</td>
<td>$9.2 \times 10^{-3}$</td>
<td>$1.8 \times 10^{-5}$</td>
</tr>
<tr>
<td>Control. Impl.</td>
<td>32-b $\rightarrow$ 16-b</td>
<td>$5.2 \times 10^{-4}$</td>
<td>$2.9 \times 10^{-4}$</td>
</tr>
<tr>
<td>Diff. image filter (5x5)</td>
<td>32-b $\rightarrow$ 16-b</td>
<td>$1.2 \times 10^{-2}$</td>
<td>$2.5 \times 10^{-3}$</td>
</tr>
<tr>
<td>FFT (N=8)</td>
<td>32-b $\rightarrow$ 16-b</td>
<td>$8.1 \times 10^{-2}$</td>
<td>$4.4 \times 10^{-3}$</td>
</tr>
<tr>
<td>IFFT (N=8)</td>
<td>32-b $\rightarrow$ 16-b</td>
<td>$8.4 \times 10^{-2}$</td>
<td>$3.2 \times 10^{-2}$</td>
</tr>
</tbody>
</table>

If we reduce microcontroller’s bit-width, how much error will be introduced?
## Experiment *(runtime statistics)*

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Optimized Code Regions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sobel image filter</td>
<td>22</td>
<td>2s</td>
</tr>
<tr>
<td>Bicycle controller</td>
<td>2</td>
<td>5s</td>
</tr>
<tr>
<td>Locomotive controller</td>
<td>1</td>
<td>5m 41s</td>
</tr>
<tr>
<td>IDCT (N=8)</td>
<td>3</td>
<td>2.7s</td>
</tr>
<tr>
<td>Controller impl.</td>
<td>1</td>
<td>46s</td>
</tr>
<tr>
<td>Differ. image filter</td>
<td>23</td>
<td>10s</td>
</tr>
<tr>
<td>FFT (N=8)</td>
<td>14</td>
<td>1m 9s</td>
</tr>
<tr>
<td>IFFT (N=8)</td>
<td>1</td>
<td>4s</td>
</tr>
</tbody>
</table>
Conclusions

• We presented a new SMT-based method for optimizing fixed-point linear arithmetic computations in embedded software code
  – Effective in reducing the required bit-width
  – Scalable for practice use

• Future work
  – Other aspects of the performance optimization, such as execution time, power consumption, etc.
More on Related Work

• Solar-Lezama et al. Programming by sketching for bit-streaming programs, *ACM SIGPLAN’05*.  
  – General program synthesis. Does not scale beyond 3-4 LoC for our application.

• Gulwani et al. Synthesis of loop-free programs, *ACM SIGPLAN’11*.  
  – Synthesizing bit-vector programs. Largest synthesized program has 16 LoC, taking >45mins. Do not have incremental optimization.

  – Computing the minimal required bit-width for fixed-point representation. Do not change the code structure.

• Rupak et al. Synthesis of minimal-error control software, *EMSOFT’12*.  
  – Synthesizing fixed-point computation from floating-point computation. Again, only compute minimal required bit-widths, without changing code structure.