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

Hassan Eldib and Chao Wang
Department of ECE, Virginia Tech, Blacksburg, VA 24061, USA
E-mail: {heldib,chaowang}@vt.edu

***Abstract*—We present a new method for optimizing the C/C++ code of embedded control software with the objective of minimizing implementation errors in the linear fixed-point arithmetic computations caused by overflow, underflow, and truncation. Our method relies on the use of an SMT solver to search for alternative implementations that are mathematically equivalent but require a smaller bit-width, or implementations that use the same bit-width but have a larger error-free dynamic range. Our systematic search of the bounded implementation space is based on an inductive synthesis procedure, which guarantees to find a solution as long as such solution exists. Furthermore, the synthesis procedure is applied incrementally to small code regions – one at a time – as opposed to the entire program, which is crucial for scaling the method to programs of realistic size and complexity. We have implemented our method in a software tool based on the Clang/LLVM compiler and the Yices SMT solver. Our experiments, conducted on a set of representative benchmarks from embedded control and DSP applications, show that the method is both effective and efficient in optimizing fixed-point arithmetic computations in embedded software code.**

I. Introduction

Analyzing and optimizing the fixed-point arithmetic computations in embedded control software is crucial to avoid overflow and underflow errors and minimize truncation errors within the designated input range. Implementation errors such as overflow, underflow, and truncation can lead to degradation of the computation results, which in turn may destabilize the entire system. The conventional solution is to carefully estimate the minimum bit-width required by the software code to run in the error-free mode and then choose a microcontroller that matches the minimum bit-width. However, this can be expensive or even infeasible, e.g., when the microcontroller at hand has 16 bits but the code requires 17 bits.

In many cases, it is possible for the developer to manually reorder the arithmetic operations to avoid the overflow and underflow errors and to minimize the truncation errors. However, the process is labor intensive and error prone. In this paper, we present a new compiler assisted code transformation method to automate the process. More specifically, we apply inductive synthesis incrementally to optimize the arithmetic computations so that the code can be safely executed on microcontrollers with a smaller bit-width.

Consider the code in Fig. 1, where all input parameters are assumed to be in the range $[0, 9000]$. A quick analysis of this program shows that, to avoid overflow, the program must be executed on a microcontroller with at least 32 bits. If it were to run on a 16-bit microcontroller, many of the arithmetic operations, e.g., the subtraction in Line 13, would overflow. In this case, a naive solution is to scale down the bit-widths of the overflowing operations by eliminating some of their least significant bits (LSBs). However, it would decrease the dynamic range, ultimately leading to a large accumulative error in the output.

Our method, in contrast, can reduce the minimum bit-width required to run this fixed-point arithmetic computation code without loss in accuracy. It takes the original C code in Fig. 1 and the user-specified ranges of its input parameters, and returns the optimized C code in Fig. 2 as output. Our method guarantees that the two programs are mathematically equivalent but the one in Fig. 2 requires a smaller bit-width. More specifically, the new code can run on a 16-bit microcontroller. Furthermore, our method ensures that the new code does not introduce additional truncation errors.

The optimization in our method is carried out by an SMT solver based *inductive synthesis* procedure, which is customized specifically for efficient handling of fixed-point arithmetic computations. Recent years have seen a renewed interest in applying inductive synthesis techniques to a wide variety of applications (e.g., [1], [2], [3], [4], [5], [6], [7], [8]). However, a naive application of inductive synthesis to our problem would not work, due to the limited scalability and large computational overhead of the synthesis procedure. For example, our experience with the Sketch tool [1] shows that, for synthesizing arbitrary fixed-point arithmetic computations, it does not scale beyond programs with 3-4 lines of code.

Our main contribution in this paper is a new *incremental inductive synthesis* algorithm, where the SMT solver based analysis is carried out only on small code regions of bounded size, one at a time, as opposed to the entire program. This incremental optimization approach allows our method to scale up to programs of practical size and complexity.

Our new method differs from existing methods for optimizing arithmetic computations in embedded software. These existing methods, including recent ones [9], [10], focus primarily on computing the optimal (smallest) bit-widths for the program variables. Instead, our method focuses on re-ordering the arithmetic operations and re-structuring the code, which in turn may lead to reduction in the minimum bit-width. In other words, we are not merely *finding* the minimum bit-width, but also *reducing* it through proper code transformation. Due to the use of an SMT solver based exhaustive search, our method can find the best implementation solution within a bounded search space.

We have implemented our method in a software tool based on the Clang/LLVM compiler framework [11] and the Yices SMT solver [12]. We have evaluated its performance on a representative set of public domain benchmarks collected from embedded control and digital signal processing (DSP) applications. Our results show that the new method can significantly reduce the minimum bit-width required by the program and,

```
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:}
```

Fig. 1. The original C program for implementing an embedded controller.

```
int comp(int A,int B,int H,int E,int D,int F,int K) {
 int t0,t1,t3,t4,t5,t6,t8,t12;
 int N1,N2,N3,N4,N5,N6,N7,N9,N10;
 t12 = 3 * A;
 N6  = H;
 N10 = t12 - B;
 N9  = N10 >> 1;
 N7  = B + N9;
 N5  = N7 >> 1;
 N4  = N5 + N6;
 t6  = N4 >> 1;
 t8  = 3 * E;
 N3  = t8 - 16469;
 t5  = N3 + D;
 t3  = t5 + t6;
 t4  = 12 * F;
 N2  = t4 >> 2;
 N1  = t3 >> 2;
 t1  = N1 - N2;
 t0  = t1 + K;
 return t0;
}
```

Fig. 2. Optimized C code for implementing the same embedded controller.

alternatively, increase the error-free dynamic range.

To sum up, this paper makes the following contributions:

- We propose the first method for incrementally optimizing the linear fixed-point arithmetic computations in embedded C/C++ code via inductive synthesis to reduce the minimum bit-width and increase the dynamic range.
- We implement the new method in a software tool based on Clang/LLVM and the Yices SMT solver, and demonstrate its effectiveness and scalability on a set of representative embedded control and DSP applications.

The remainder of this paper is organized as follows. In Section II, we illustrate our new method by using an example. We establish the notation in Section III, and then present the overall algorithm in Section IV. We present our inductive synthesis procedure in Section V. The implementation details and experimental results are given in Section VI and Section VII, respectively. We review related work in Section VIII, and finally give our conclusions in Section IX.

II. Motivating Example

We illustrate the overall flow of our method using the example in Fig. 1. The program is intended to be simple for ease of presentation. In the actual evaluation benchmarks, the programs have loops and variables that are assigned more than once. Note that loops in these programs are all bounded, and therefore can be handled by finite unrolling.

Our method takes the program in Fig. 1 and a configuration file that defines the value ranges of all parameters as input, and returns the program in Fig. 2 as output. It starts by parsing the original program and constructing an abstract syntax tree (AST). Each variable in Fig. 1 corresponds to a node in the AST. The root node is the return value of the program. The leaf nodes are the input parameters.

The AST is first traversed *forwardly*, from the parameters to the return value, to compute the value ranges. Each value range is a (min, max) pair for representing the minimum and maximum values of the node, computed using a symbolic range analysis [13]. Then, the AST is traversed *backwardly*, from the return value to the parameters, to identify the list of AST nodes that may overflow or underflow when using a reduced bit-width. For example, the first overflowing node in Fig. 1 is the subtraction in Line 13: although `t3` and `t4` can be represented in 16 bits, the subtraction may produce a value that requires more bits.

For each AST node that may overflow or underflow, we carve out some neighboring nodes to form a *region for optimization*. The region includes the node, its parent node, its child nodes, and optionally, the transitive fan-in and fan-out nodes up to a bounded depth. The region size is limited by the capacity of the inductive synthesis procedure. For the subtraction in Line 13, if we bound the region size to 2 AST levels, the extracted region would include the right-shift in Line 14, which is the parent node.

The region is then subjected to an inductive synthesis procedure, which generates an equivalent region that does not overflow or underflow. For Line 13 in Fig. 1, the extracted region and the new region are shown side by side as follows:

```
t2 = t3 - t4;                   N2 = t4 >> 2;
t1 = t2 >> 2;         -->       N1 = t3 >> 2;
...                             t1 = N1 - N2;
```

That is, instead of applying right-shift to the operands after subtraction, it applies right-shift first. Because of this, the new region needs a smaller bit-width to avoid overflow.

However, the above new region is not always better because it may introduce additional truncation errors. Consider `t3 = 2, t4 = -2` as a test case. We have `(t3 - t4) >> 2 = 1` and `(t3 >> 2 - t4 >> 2) = 0`, meaning that the new region may lose precision if the two least significant bits (LSBs) of `t3,t4` are not zero. An integral part of our new synthesis method is to make sure that the new region does not introduce additional truncation errors. More specifically, we perform a truncation error margin analysis to identify, for each AST node, the number of LSBs that are immaterial in deciding the final output. For Line 13, this analysis would reveal that the LSBs of `t3` and `t4` do not affect the value of the final output.

Since the new region is strictly better, the original AST is updated by replacing the extracted region with the new region. After that, our method continues to identify the next node that may overflow or underflow. The entire procedure terminates when it is no longer possible to optimize any further.

In the remainder of this section, we provide a more detailed description of the subsequent optimization steps.

After optimizing the subtraction in Line 13, the next AST node that may overflow is in Line 10. The extracted region and the new region are shown side by side as follows:

```
t7 = t8 + D;                    N3 = t8 - 16469;
```

```
t5 = t7 - 16469;        -->        t5 = N3 + D;
```

Our analysis shows that variables `t8, D` and constant `16469` all have zero truncation error margins. The new region does not introduce any additional truncation error. Therefore, the original AST is updated with the new region.

The next AST node that may overflow is in Line 6. The extracted region and the new region are shown as follows:

```
t9 = t10 + t11;                   N6 = t11 >> 2;
t6 = t9 >> 3;                     N5 = t10 >> 2;
...                     -->       N4 = N5 + N6;
...                               t6 = N4 >> 1;
```

The truncation error margins are 2 for `t10` and 2 for `t11`. Therefore, the truncation error margin for `t9` is 2, meaning that the two LSBs may be ignored. Since the new region is strictly more accurate, the original AST is again updated with the new region.

The next AST node that may overflow is in Line 4. The extracted region and the new region are shown as follows:

```
t10 = t12 + B;                    N10 = t12 - B;
N5  = t10 >>2;                    N9  = N10 >> 1;
...                     -->       N7  = B + N9;
...                               N5  = N7 >> 1;
```

Notice that this extracted region consists of a node that is the result of a previous optimization step. The truncation error margins are 0 for `t12` and 0 for `B`. The new code region does not suffer from the same truncation error that would be introduced by `N5 = (B>>2 + t12>>2)`, because the truncation error is not amplified while being propagated to the final result. Instead, it is compensated by the addition of `B`.

The last node that may overflow is in Line 5 of Fig. 1. The extracted region and the new region are shown as follows:

```
t11 = H << 2;
N6  = t11 >> 2;         -->        N6 = H;
```

By now, all arithmetic operations that may overflow are optimized. The new program in Fig. 2 can run on a 16-bit microcontroller while still maintaining the same accuracy as the original program running on a 32-bit microcontroller. Another way to look at it is that if the optimized code were to be executed on the original 32-bit microcontroller, it would have a significantly larger dynamic range.

III. Preliminaries

A. Fixed-point Notations

We follow [14] to represent the fixed-point type by a tuple $\langle s, N, m\rangle$, where s indicates whether it is signed or unsigned (1 for signed and 0 for unsigned), N is the total number of bits or the *bit-width*, and m is the number of bits representing the fractional part. The number of bits representing the integer part is $n = (N - m)$. Different variables and constants in the original program are allowed to have different bit representations, but all of them should have the same bit-width N.

Signed numbers are represented in the standard *two's complement* form. For an N-bit number α, which is represented by bit-vector $x_{N-1}\ x_{N-1}\ ...\ x_0$, its value is defined as follows:

$$\alpha = \frac{1}{2^m} \times \left(-2^{N-1} x_{N-1} + \sum_{i=0}^{N-2} 2^i x_i \right) \ ,$$

where x_i is the value of the i^{th} bit. The value of α lies in the range $[-2^n, 2^n - 2^{-m}]$. If a number to be represented exceeds the maximum value, there will be an *overflow*. If a number to be represented is less than the minimum value, there will be an *underflow*. If the number to be represented requires more designated fractional bits than m, there will be a *truncation error*. The maximum error caused by truncation is 2^{-m}.

We define the *step* of a variable or a constant as the number of consecutive LSBs that always have the value zero. For example, the number 1024 has a *step* 9, meaning that nine of the LSBs are zero. On the other hand, the number 3 has a *step* 0. During the optimization process, they will be used to compute the truncation error margin (the LSBs whose values can be ignored). Our method will leverage the truncation error margins to obtain the best possible optimization results.

B. Intermediate Representation

We use Clang/LLVM to construct an intermediate representation (IR) for the input program. Since the standard C language cannot explicitly represent fixed-point arithmetic operations, we use a combination of the integer C program representation and a separate configuration file, which defines the fixed-point types of all program variables. More specifically, we scale each fixed-point constant (other than the ones used in *shift* operations) to an integer by using the scaling factor 2^m. For example, a constant with the value of 2.5 will be represented as 10, together with $m = 2$, since $2.5 * 2^2 = 10$.

After each multiplication, a *shift-right* is added to normalize the result so as to match the fixed-point type for the result. For example, $x = c \times z$, where variables x and z and constant c all have the fixed-point type $\langle 1, 8, 3\rangle$, would be represented as $x = (c \times z) >> 3$. Our implementation currently supports linear fixed-point arithmetic only; therefore, we do not consider the multiplication of two variables.

Although there is no inherent difficulty in our method for handling non-linear arithmetic, we focus on linear arithmetic for two reasons. First, the benchmarks used in our experiments are all linear. Second, we have not evaluated the efficiency of SMT solvers in handling non-linear arithmetic operations. Therefore, we leave the handling of nonlinear arithmetic for future work.

For each multiplication, we also assign an *accumulate flag*, which can be set by the user to indicate whether the microcontroller has the capability of temporally storing the multiplication result into two registers, which effectively doubles the bit-width of the registers. Many real-world microcontrollers have been designed in this way. Continuing with the same example $x = (c \times z) >> 3$, if the *accumulate flag* is set to 1 by the user, the multiplication node will not be checked for overflow and underflow. Only after the right-shift, will the final result be checked for overflow and underflow.

For all the other operations (`+, -, >>, <<`), we do not rewrite the default IR representation and do not allow the user to set the *accumulate flag*, because most of the microcontrollers do not have double sized registers to temporarily store the results of these operations.

IV. The Overall Algorithm

The overall flow of our method in shown in Algorithm 1. The input includes the original program and the value ranges of all the parameter variables. First, we invoke COMPUTERANGES to compute the value ranges of all non-leaf AST nodes. Then, we invoke COMPUTEIGNOREBITS to compute the truncation error margins (LSBs whose values can be ignored) for all AST nodes. Finally, we compute the bit-width ($bw1$) required by the original program to run within the given input range.

Algorithm 1 Optimizing the program within its input range.

```
1: OPTIMIZEPROGRAM ($prog, p\_ranges$) {
2:   $ranges \leftarrow$ COMPUTERANGES($prog$,$p\_ranges$);
3:   $ig\_bits \leftarrow$ COMPUTEIGNOREBITS($prog$);
4:   $bw1 \leftarrow$ COMPUTMINBITWIDTH($prog$,$ranges$);
5:   while (true) {
6:     $bw2 \leftarrow bw1 - 1$;
7:     for each (Node $n \in prog$ that may overflow or underflow) {
8:       $reg \leftarrow$ EXTRACTREGION($prog$,$n$);
9:       $new\_reg \leftarrow$ SYNTHESIZE($reg$,$bw1$, $bw2$, $ranges$, $ig\_bits$);
10:      if ($new\_reg$ does not exist) break;
11:      REPLACEREGION($prog$,$reg$, $new\_reg$);
12:    }
13:    $bw1 \leftarrow bw2$;
14:  }
15:  return $prog$;
16: }
```

After the bit-width of the original program ($bw1$) is determined, we enter the while-loop to iteratively optimize the program. In each iteration, we try to reduce the bit-width from $bw1$ to $bw2$. The loop terminates as soon as a call to the inductive synthesis procedure fails to return the new region.

Within each loop iteration, we search for all nodes that may overflow or underflow when the new bit-width ($bw2$) is used. We process these nodes in a breadth-first search (BFS) order, i.e., from the return value of the program to the parameter variables. For each node, we invoke EXTRACTREGION to extract a neighboring region and then invoke the inductive synthesis procedure. If successful, the inductive synthesis procedure would return a new region, which is mathematically equivalent to the extracted region but would not overflow or underflow. It also ensures that the new region would not introduce additional truncation error. After the new region is found, we use it to replace the extracted region in the program.

A. Region for Optimization

The size of the extracted *region* affects both the effectiveness and the computational overhead of the inductive synthesis procedure. The minimum extracted region should include the erroneous node and its parent node. Since we follow the BFS order, the parent node must have no overflow or underflow since it is already tested negative or optimized. Since in the original program, the parent operation restores the overflowed value created in the overflowing node back to the normal operation range, when the parent node is included in the region, it is more likely to find an alternative implementation that is more accurate than the extracted region.

In general, a larger extracted region allows for more opportunity to find a suitable new region. The maximum extracted region – if it were not for the limited capability of the SMT solver – would be the entire input program. This is equivalent to applying inductive synthesis tools such as Sketch [1], [2] to the entire program, provided that the fixed-point arithmetic optimization problem is modeled in the Sketch input language. In practice, however, such a monolithic optimization approach seldom works. Indeed, our experience with the Sketch tool shows that it cannot scale beyond arbitrary fixed-point arithmetic computation code of 2-3 lines.

Therefore, in addition to implementing our customized inductive synthesis procedure, which can efficiently handle fixed-point arithmetic computations, we also bound the size of the extracted region so that inductive synthesis is applied only in the context of incremental optimization. More specifically, the extracted region is bounded to an AST with at most 5 node levels, which represents up to 63 AST nodes.

B. Truncation Error Margin

We compute the *step* and the *ignore bits* for all AST nodes recursively. First, we determine the *step* of each leaf node based on the definition in Section III. In general, the *step* may originate from a *shift-left* operation, a *step* in a parameter variable, or a *step* in a constant. We compute the step of each internal AST node as follows:

- $step(x * y) = step(x) + step(y)$;
- $step(x + y) = min(step(x), step(y))$;
- $step(x - y) = min(step(x), step(y))$;
- $step(x << c) = step(x) + c$;
- $step(x >> c) = max(step(x) - c, 0)$.

The *ignore bits* are those consecutive LSBs that can be ignored during the optimization process. If these bits are truncated in the new region, for example, no error will occur in its output. By taking into account these bits in the optimization process, we are able to synthesis more compact new regions.

To clarify this, consider the example in Fig. 3, where the extracted region is shown inside the dotted box. We start by analyzing the AST to determine the *step* of each node. For the purpose of optimizing the extracted region, we need to know the *step* of the region's inputs, which are the nodes labeled as a and b. Due to the shift-left operations, the *step* of a is 4, while the *step* of b is 3. Considering these *step* values, we determine that, when optimizing the extracted region, we have a "credit" of 3 bits to ignore. In other words, we have the freedom to truncate up to 3 consecutive LSBs of the two inputs (a and b) without decreasing the accuracy of the result. Because of this, we are able to synthesize the new region as shown in Fig. 4.

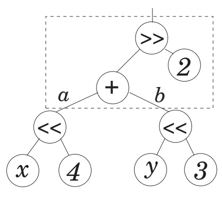

Fig. 3. The extracted region.

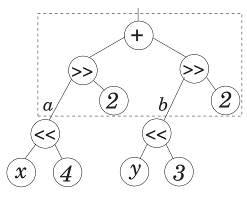

Fig. 4. The synthesized region.

Notice that, even if we do not consider the ignore bits, our method can still synthesize a new region to remove the overflowing node in the above example. However, in such case, the extracted region would have to be larger. That is, the extracted region would need to include all the AST nodes in Fig. 3. The synthesized new region would include all the AST nodes in Fig. 4. However, this would also lead to a significantly longer synthesis time.

V. The Inductive Synthesis Procedure

At the high level, our inductive synthesis procedure consists of two steps: (1) run a set of test cases on the extracted region, and based on the results, generate a new region that is equivalent to the extracted region at least for the set of test cases; (2) check if the two regions are equivalent in the full input range. If they are not equivalent, block this region (bad solution) and try again.

Algorithm 2 shows the pseudo code of our synthesis procedure, which computes a new region (new_reg) of bit-width $bw2$, such that it is equivalent to the original region (reg) of bit-width $bw1$, under the value ranges specified in $ranges$ while considering the truncation error margins specified in ig_bits. The procedure starts by initializing *blockedRegions* and *testSet* to empty sets, where *testSet* consists of the test cases used for inductively generating (guessing) a new region, and *blockedRegions* consists of the previously explored regions that fail the equivalence check. The procedure initializes the *size* of the new region to 1, and then enters the while-loop to iteratively search for a new region of increasingly larger size. When *size* exceeds a predetermined bound, we have proved that no solution exists in this search space.

Subroutine genRegion uses an SMT solver to inductively generate a new region, based on the test examples in *testSet* and the already explored regions in *blockedRegions*. Subroutine compDiff formally checks the equivalence of the extracted region (reg) and the new region (new_r), and returns a concrete test if they are not equivalent.

Algorithm 2 Inductively synthesizing the new code region.

```
1:  SYNTHESIZE ($reg, bw1, bw2, ranges, ig\_bits$) {
2:    blockedRegions ← { };
3:    testSet ← { };
4:    $size \leftarrow 1$;
5:    while ($size <$ MAX_REGION_SIZE) {
6:      new_r ← GENREGION($reg, bw1, bw2, size$, blockedRegions, testSet);
7:      if ($new\_r$ exists) {
8:        $test$ ← COMPDIFF($reg, new\_r, bw1, bw2, ranges, ig\_bits$);
9:        if ($test$ exists) {
10:         blockedRegions ← blockedRegions $\cup\{new\_r\}$;
11:         testSet ← testSet $\cup\{test\}$;
12:       }
13:       else
14:         return $new\_r$;
15:     }
16:     else
17:       $size \leftarrow size + 1$;
18:   }
19:   return no_solution;
20: }
```

A. Constructing the New Region Skeleton

First, we generate a *skeleton* of the new region, which is a generalized AST capable of representing any linear arithmetic equation up to a bounded size. In this AST, each leaf node is either a constant or any of the set of input variables of the extracted *region*. Each internal node is any of the linear arithmetic operations (`*, +, -, >>, <<`). The root node is the result of the arithmetic computation and should compute the same result as the output node in the extracted region. Fig. 5 shows an example *skeleton* of 7 AST nodes. Here, *Op* represents any binary arithmetic operator and $V|C$ represents a leaf node (either a variable or a constant).

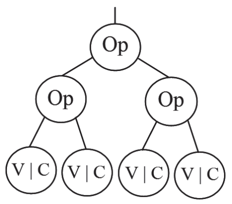

Fig. 5. Skeleton of 7 AST nodes.

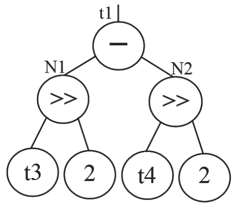

Fig. 6. Synthesized new region.

For each AST node in the *skeleton*, we assign an auxiliary variable called the *selector*, whose value determines the node type. For example, a leaf node (`LNode1`), which may be variable `V1`, variable `V2`, or constant `C1`, is represented as follows:

```
((LNode1 == V1) && (sel1 == 0) ||
 (LNode1 == V2) && (sel1 == 1) ||
 (LNode1 == C1) && (sel1 == 2))
```

where the integer value of selector variable `sel1` ranges from 0 to 2. Similarly, a generalized internal node (`INode3`), which may be an addition or a subtraction of `LNode1` and `LNode2`, is represented as follows:

```
((INode3 == LNode1+LNode2) && (sel2 == 0) ||
 (INode3 == LNode1-LNode2) && (sel2 == 1))
```

where the integer value of selector variable `sel2` ranges from 0 to 1. The actual node types in the *skeleton* are determined later, when we encode the skeleton into an SMT formula, and then call the SMT solver to find a set of suitable values for all these selector variables.

B. Inductively Generating the New Region

To generate the new region, we need a representative set of test cases for the extracted region. These are test values for the input variables of the region, and should include both the corner cases and the intermediate values. Since the arithmetic computations are linear, we construct the corner cases by including the minimum and maximum values of all input variables as defined in *ranges*. Additional test values are generated by taking semi-equidistant intermediate values between values in the corner cases.

We create an SMT formula Φ such that Φ is satisfiable iff the resulting new region – induced by a satisfying assignment to all *selector* variables – is mathematically equivalent to the extracted region, but does not overflow or underflow.

$$\Phi = \Phi_{reg} \land \Phi_{skel} \land \Phi_{sameI} \land \Phi_{sameO} \land \Phi_{tests} \land \Phi_{blocked},$$

where the subformulas are defined as follows:

- Extracted region (Φ_{reg}): It encodes the transition relation of the extracted region by using bit-vector arithmetic, where the bit-width is *bw1*.
- New region skeleton (Φ_{skel}): It encodes the transition relation of the skeleton by using bit-vector arithmetic, where the bit-width is *bw2*.
- Same input values (Φ_{sameI}): It asserts that the input variables of the two regions must share the same values.
- Same output value (Φ_{sameO}): It asserts that the output variables of the two regions must have the same value, and there is no overflow or underflow.

- Test cases (Φ_{tests}): It asserts that the input variables must adopt concrete values from the given test cases.
- Blocked solutions ($\Phi_{blocked}$): It asserts that the *selector* variables should not take values that represent any previously explored (bad) solution.

If Φ is unsatisfiable, no solution exists in the bounded search space. In this case, we need to increase the *size* of the *skeleton* and try again. If Φ is satisfiable, we have computed a candidate new region. As an example, consider the first extracted region in Section II. The new region generated from the skeleton in Fig. 5 is shown in Fig. 6.

C. Checking the Equivalence of the Regions

The candidate new region is guaranteed to be equivalent to the extracted region over the given set of test cases. However, they may not be equivalent over the full input range. Therefore, the next step is to formally verify their equivalence over the full input range. Toward this end, we create another SMT formula Ψ, which is satisfiable iff the two regions are *not* equivalent; that is, if there exists a test case that can differentiate them. Formula Ψ is defined as follows:

$$\Psi = \Phi_{reg} \wedge \Phi_{new\_reg} \wedge \Phi_{sameI} \wedge \Phi_{diffO} \wedge \Phi_{ranges} \wedge \Phi_{ig\_bits},$$

where the subformulas are defined as follows:

- New region (Φ_{new_reg}): It encodes the transition relation of the candidate new region in bit-vector arithmetic, where the bit-width is *bw2*.
- Different output values (Φ_{diffO}): It asserts that the output variables of the two regions have different values.
- Value ranges (Φ_{ranges}): It asserts that all input variables should stay within their pre-computed value ranges. We are not interested in checking the equivalence of the two regions outside the designated value ranges.
- Ignore bits (Φ_{ig_bits}): It asserts that the LSBs as specified in the ignore bits should all be set to zero. This allows us to ignore the differences between the two regions for LSBs within the truncation error margins.

If Ψ is unsatisfiable, it means that the two regions are mathematically equivalent within the given input range and under the consideration of the ignore bits.

If Ψ is satisfiable, the candidate new region is not correct. In this case, we add it to the *blockedRegions* and try again. The blocking of an incorrect solution follows the counter-example guided inductive synthesis algorithm [1], [15], where the blocked solutions are encoded as an additional constraint in the SMT formula, by adding an extra pair of extracted *region* and new region *skeleton* with the blocked assignment to *selector* variables. It ensures that, when the SMT solver is invoked again to find a candidate new region, the same solution will not be returned.

VI. Implementation

We have implemented our new method in a software tool for optimizing the C/C++ code of embedded control and DSP applications based on the Clang/LLVM compiler framework [11] and the Yices SMT solver [12]. Our tool has two modes: the whole-program optimization mode and the incremental optimization mode. The two modes differ only in the size bound imposed on the extracted region.

When the bound is set to an arbitrarily large number, our tool runs in the whole-program optimization mode. This makes it somewhat comparable to the popular inductive synthesis tool called Sketch [1], [15], provided that our new region *skeleton* is carefully modeled in the Sketch input language, with the *selector* variables defining the “integer holes” for Sketch to fill. Before implementing our own inductive synthesis procedure, we have explored this approach. However, it turns out to be not scalable: synthesizing a new region with a size bound of more than 2 would cause Sketch to quickly run out of the 4 GB memory. We believe that there are two reasons for this. First, the performance of Sketch is not optimized for handling arbitrary combinations of linear fixed-point arithmetic computations. Second, inductive synthesis, in general, may not be able to scale up to arbitrarily large arithmetic computation programs.

Due to the scalability problem encountered by using Sketch, we have implemented our own inductive synthesis procedure directly using the Yices SMT solver, which is designed for efficient handling of fixed-point arithmetic operations, e.g., by designing SMT encoding schemes for exploiting the AST structures encountered in this type of applications. Our experimental evaluation shows that the new procedure is significantly more efficient than Sketch. Instead of a size bound of 2, it now can routinely optimize the *skeleton* with a size bound of 5 (representing up to 63 AST nodes). Nevertheless, this improvement alone is not sufficient for supporting the whole-program optimization.

Instead, we propose an incremental optimization method that applies inductive synthesis only to individual regions of a bounded size. More specifically, we have set the maximum bound for *shift-right* and *shift-left* operations to 4, and the maximum level of AST nodes in the new region skeleton to 5. By incrementally optimizing one extracted region at a time, our method is able to avoid the scalability bottleneck imposed by the SMT solver, and therefore can be applied to programs of practical size and complexity.

VII. Evaluation

We have evaluated our tool on a set of public domain benchmark examples. The experiments are designed to answer the following three questions:

- How much can our method reduce the minimum bit-width required for the program to run in the given input range?
- How much can our method increase the dynamic range of the program for the given bit-width?
- If both the original and the optimized programs are forced to run with a reduced bit-width, what is the difference between their fixed-point specific implementation errors?

A. Benchmarks

Our benchmark includes a set of public domain C programs for embedded control and DSP applications. They come from various sources including papers, textbooks, and the output of code generation tools. The sizes of the programs range from 21 lines of code (LoC) to 131 lines, with an average LoC of 79. The number of fixed-point arithmetic operations on average is 58. For the kind of cyber-physical systems (CPS) software targeted by our new method, these are programs of realistic size and complexity.

TABLE I
STATISTICS OF THE BENCHMARK C PROGRAMS.

Name of the Benchmark	Line of Code	Arithmetic Operations
Sobel Image filter (3x3)	42	28
Bicycle controller	37	27
Locomotive controller	42	38
IDCT (N=8)	131	114
Control. Impl.	21	8
Diff. image filter (5x5)	131	77
FFT (N=8) (no DC component)	112	82
IFFT (N=8)	112	90

TABLE II
INCREASE IN THE OVERFLOW/UNDERFLOW FREE INPUT RANGE.

benchmark	bit	original	optimized	%
Sobel Image	32	[0, 16320]	[-65536, 49152]	602
Bicycle	32	[-3.4*10^8, 3.4*10^8]	[-1.0*10^9, 1.0*10^9]	194
Locomotive	64	[-8.7*10^{18}, 8.7*10^{18}]	[-9.2*10^{18}, 9.2*10^{18}]	5
IDCT	32	[0, 1.5*10^6]	[0, 2.1*10^6]	40
Controller	32	In1 [0, 5.0*10^8]	In1 [-0, 6.6*10^8]	32
		In2 [-5.0*10^8, 0]	In2 [-6.6*10^8, 0]	32
		In3 [-5.0*10^8, 0]	In3 [-6.6*10^8, 0]	32
Diff. Image	32	[0, 1.3*10^8]	[0, 2.1*10^9]	1515
FFT (N=8)	32	[0, 32736]	[0, 32736]	0
IFFT (N=8)	32	[0, 2.6*10^8]	[0, 5.3*10^8]	103

TABLE III
INCREASE IN THE MINIMUM AND AVERAGE BIT-WIDTHS.

Name of	Original (bit-width)		Optimized (bit-width)	
Benchmark	Minimum	Average	Minimum	Average
Sobel image filter (3x3)	17	10.26	15	6.67
Bicycle controller	18	14.47	16	14.16
Locomotive controller	33	29.41	32	29.32
IDCT (N=8)	20	16.29	19	16.38
Control. Impl.	17	15	16	14.67
Diff. image filter (5x5)	17	11.11	13	8.09
FFT (N=8)	18	7.32	16	6.95
IFFT (N=8)	17	7.11	16	7.26

Table I shows the statistics of our benchmarks. The first test case, taken from [16], is a 3x3 Sobel digital filter that is widely used in image processing applications. The second test case, taken from [10], is a bicycle controller optimally synthesized for a custom-designed microprocessor with double-sized internal registers. The third test case is a locomotive controller generated by using Fixed Point Advisor and Real Time Workshop of the Matlab toolkit [17]. The fourth test case, taken from [18], is an inverse discrete cosine transform (IDCT), which is widely used in mobile communication and image compression applications. The fifth test case is the fixed-point version of a control rule implementation from [17]. The sixth test case is a 5x5 kernel sized difference image filter [19]. The seventh test case is a fast Fourier transform (FFT) implementation, where the floating-point version was taken from [20] and then converted to fixed-point, by changing all `double` variables into `int` variables without modifying or reordering any of its instructions. The eighth test case is the inverse fast Fourier transform (IFFT) for test case 7. None of the benchmarks was modified from their original forms in any way to give performance advantage to our method.

All experiments were conducted on a machine with a 3.4 GHz Intel i7-2600 CPU, 3.3GB of RAM, and 32-bit Linux.

B. Results

First, we show that there is a significant increase in the input/output range from the original program to the optimized program, when they both use the original bit-width. Table II shows the results (data on the output range are similar, and therefore are omitted for brevity). Column 1 shows the name of the benchmark. Columns 2 and 3 show the input (output) ranges of the original program and the optimized program, respectively. Column 4 shows the percentage of the range increase. The increase in input (output) range spans from 0% to 1515%, with an average of 307% or a median of 72%. The increase is due to the removal of the overflowing and underflowing nodes in the original program. As a result, the output range is also increased. Together, they lead to a significant increase in the dynamic range of the entire application.

Second, we show that there is a significant decrease in the minimum bit-width required for the program to run without overflow/underflow errors for the given input range. The experimental results are shown in Table III. Column 1 is the name of the benchmark. Column 2 is the minimum bit-width of the original program to avoid overflow and underflow, and Column 3 is the average bit-width for all program variables. Column 4 is the minimum bit-width of the new program to avoid overflow and underflow, and Column 5 is the average bit-width for all program variables.

Our results show that the bit-width reduction spans from 1 bit to 4 bits. Consider the Sobel Image filter as an example. The minimum bit-width required to run the original program is 17 bits. After optimization, it is reduced to 15 bits. This is significant, because now the code can be executed on a 16-bit microcontroller instead of a 32-bit microcontroller, which is often significantly cheaper.

To further illustrate the benefit of our new method, consider the maximum error bound in a scaled-down version of the original program in order to downgrade the hardware from 32-bit to 16-bit, or from 64-bit to 32-bit. Table IV shows the comparison between the optimized program and a scaled-down version of the original program. Column 1 is the name of the benchmark. Column 2 is the scaling level. Columns 3 and 4 are the maximum relative errors of the original program and the optimized program, respectively. Our results show that the optimized programs have smaller errors in all test cases.

We also show, in Table V, the statistics of running our optimization method. Column 1 is the name of the benchmark. Column 2 is the number of lines optimized by the incremental inductive synthesis procedure in the original program. Column 3 is the total execution time by our method. The data show that, by using incremental synthesis, we have kept the overall runtime down. In fact, it is no longer directly dependent on the

TABLE IV
DECREASE IN THE MAXIMUM RELATIVE ERROR.

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

TABLE V
STATISTICS OF THE INCREMENTAL OPTIMIZATION PROCESS.

Name of the Benchmark	Num. Optimized Lines	Total Time
Sobel Image filter (3x3)	22	2s
Bicycle controller	2	5s
Locomotive controller	1	5m 41s
IDCT (N=8)	3	2.7s
Control. Impl.	1	46s
Diff. image filter (5x5)	23	10s
FFT (N=8)	14	1m9s
IFFT (N=8)	1	4s

program size, but more on the number of extracted regions and the time spent on optimizing each region. For *Locomotive*, the SMT solver took a longer time because of its larger original bit-width (64-bit) – the other examples are all 32-bit.

VIII. RELATED WORK

Our new method incrementally optimizes the fixed-point arithmetic computations in an embedded software program with the objective of reducing the minimum bit-width through code transformation, without changing the computational accuracy. The core synthesis routine in our method follows the same counter-example guided inductive program synthesis paradigm pioneered by Sketch [1], [2]. However, our method is significantly different in that it has an implementation that is designed for more efficiently handle linear fixed-point arithmetic computations. Furthermore, we apply inductive synthesis incrementally to regions of a bounded size, one at a time, as opposed to the entire program.

Gulwani *et al.* [5] propose a method for synthesizing bit-vector programs from a linear reference code by leveraging a set of user defined library functions. Their method does not use incremental inductive synthesis, and the largest synthesized code reported in their paper has 16 lines of code, for which their tool takes over 45 minutes. Jha *et al.* [3] use the same symbolic encoding as in [5] but replace the logical specification of the desired program by an input-output oracle.

The SCIDUCTION tool implemented by Jha [9] can automatically synthesize a fixed-point arithmetic program from the floating-point arithmetic code. However, the focus of this tool is solely on *finding* the smallest possible bit-width and *choosing* the best fixed-point representation for each program variable. They have not attempted to change the code structure or synthesize completely new code for the purpose of *reducing* the minimum bit-width.

Another closely related work is the linear fixed-point optimization method proposed in [10], which relies on using a Mixed Integer Linear Programming (MILP) solver to minimize the error bound by changing the fixed-point representation of the program. Again, their method can only optimize the bit-vector representations of the program variables, but do not change the structure of the original code or synthesize new completely new code in order to reduce the bit-width.

Our method is also related to superoptimization in modern compilers [21], [22], [23], which perform exhaustive search in the space of valid instruction sequences to optimize performance-critical inner loops. However, they typically cannot be used to increase the dynamic range, or minimize the bit-width, of fixed-point arithmetic computations.

IX. CONCLUSIONS

We have presented a new method for incrementally optimizing the linear fixed-point arithmetic computations of an embedded software program via code transformation to reduce the required bit-width and to increase the dynamic range. Our method is based on judicious application of an SMT solver based inductive synthesis procedure to code regions of bounded size. We have implemented our method in a software tool and evaluated it on a set of representative embedded programs. Our results show that the new method can significantly reduce the bit-width and handle programs of realistic size and complexity.

X. ACKNOWLEDGMENTS

This work is supported in part by the NSF grant CNS-1128903 and the ONR grant N00014-13-1-0527.

REFERENCES

[1] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu, "Programming by sketching for bit-streaming programs," in *PLDI*, 2005, pp. 281–294.
[2] A. Solar-Lezama, C. G. Jones, and R. Bodík, "Sketching concurrent data structures," in *PLDI*, 2008, pp. 136–148.
[3] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *ICSE*, 2010, pp. 215–224.
[4] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 2011, pp. 317–330.
[5] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *PLDI*, 2011, pp. 62–73.
[6] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *PLDI*, 2011, pp. 317–328.
[7] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *PLDI*, 2012, pp. 275–286.
[8] R. Singh and S. Gulwani, "Synthesizing number transformations from input-output examples," in *International Conference on Computer Aided Verification*, 2012, pp. 634–651.
[9] S. K. Jha, "Towards automated system synthesis using sciduction," Ph.D. dissertation, UC Berkeley, Nov 2011.
[10] M. Rupak, I. Saha, and M. Zamani, "Synthesis of minimal-error control software," in *ACM international conference on Embedded software*, 2012, pp. 123–132.
[11] C. Lattner and V. Adve, "The LLVM Instruction Set and Compilation Strategy," CS Dept., Univ. of Illinois at Urbana-Champaign, Tech. Report UIUCDCS-R-2002-2292, Aug 2002.
[12] B. Dutertre and L. de Moura, "A fast linear-arithmetic solver for DPLL(T)," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 81–94.
[13] R. Rugina and M. C. Rinard, "Symbolic bounds analysis of pointers, array indices, and accessed memory regions," in *PLDI*, 2000, pp. 182–195.
[14] R. Yates, *Fixed-point arithmetic: An introduction*. Digital Signal Labs, Technical Reference, 2013.
[15] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS*, 2006, pp. 404–415.
[16] S. Qureshi, *Embedded Image Processing on the TMS320C6000 DSP*. Springer, 2005.
[17] A. Martinez, R. Majumdar, I. Saha, and P. Tabuada, "Automatic verification of control system implementations," in *ACM international conference on Embedded software*, 2010, pp. 9–18.
[18] S. Kim, K.-I. Kum, and W. Sung, "Fixed-point optimization utility for c and c++ based digital signal processing programs," in *IEEE Trans. Circuits and Systems II*, vol. 45, no. 11, 1998, pp. 1455–1464.
[19] W. Burger and M. Burge, *Digital Image Processing*. Springer, 2008.
[20] J. Xiong, J. R. Johnson, R. W. Johnson, and D. A. Padua, "Spl: A language and compiler for dsp algorithms," in *PLDI*, 2001, pp. 298–308.
[21] R. Joshi, G. Nelson, and K. H. Randall, "Denali: A goal-directed superoptimizer," in *PLDI*, 2002, pp. 304–314.
[22] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," in *ASPLOS*, 2006, pp. 394–403.
[23] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *ASPLOS*, 2013, pp. 305–316.