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

Hassan Eldib, and Chao Wang, *Member, IEEE*

***Abstract*—We present a new method for optimizing the source 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 the satisfiability modulo theory (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 a new inductive synthesis procedure, which is guaranteed to find a valid solution as long as such solution exists. Furthermore, we propose an incremental optimization procedure, which applies the synthesis procedure only to small code regions—one at a time—as opposed to the entire program, which is crucial for scaling the method up to programs of realistic size and complexity. We have implemented our new method in a software tool based on the Clang/LLVM compiler frontend and the Yices SMT solver. Our experiments, conducted on a set of representative benchmarks from embedded control and digital signal processing applications, show that the method is both effective and efficient in optimizing arithmetic computations in embedded software code.**

***Index Terms*—Fixed point arithmetic, inductive program synthesis, satisfiability modulo theory (SMT) solver, superoptimization.**

I. Introduction

ANALYZING and optimizing the arithmetic computations in embedded control software is crucial to avoid overflow and underflow errors and minimize truncation errors within a cyber physical system's designated input range. Minimizing implementation errors is important because errors due to overflow, underflow, and truncation can lead to significant degradation of the computation results, which in turn may destabilize the entire system. The conventional solution to this problem is to carefully estimate the minimum bit-width required by all computations in the software code for them to run in the error-free mode and then choose a microcontroller that matches the minimum bit-width. However, this can be expensive in many cases and even infeasible in some cases, e.g., when the microcontroller at hand has 16 bits but the software code requires a minimum of 17 bits.

Manuscript received March 24, 2014; revised May 28, 2014; accepted July 15, 2014. Date of current version October 16, 2014. The work of H. Eldib was supported by the NSF under Grant CNS-1128903. The work of C. Wang was supported by the ONR under Grant N00014-13-1-0527. Some of the preliminary results of this paper have appeared in [1]. This paper was recommended by Associate Editor T. Mitra.

The authors are with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA 24061 USA (e-mail: heldib@vt.edu; chaowang@vt.edu).

Digital Object Identifier 10.1109/TCAD.2014.2341931

Our observation is that, in many applications, 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. In other words, one can often rewrite the software code to reduce its implementation errors while keeping the functionality of the code intact. However, the manual rewriting based optimization process is labor intensive and error prone. In this paper, we propose a new compiler assisted code transformation method to automate this optimization process. More specifically, we propose to apply a new inductive synthesis procedure incrementally to optimize the linear fixed-point arithmetic computations so that the resulting software code may be safely executed on a microcontroller with a smaller bit-width.

Consider the arithmetic computation code in Fig. 1, where the values of all input parameters are assumed to be in the range $[0, 9000]$. A static analysis of this program shows that, to avoid overflow, the program must be executed on a microcontroller with at least 32 bits. For example, if the code were to run on a 16-bit microcontroller, some of the arithmetic operations, such as the subtraction at Line 13, would overflow because the computation results cannot be represented by 16 bits. In this case, a naive solution to the problem is to scale down the bit-widths of the operations, often called binary scaling and rescaling, by eliminating some of their least significant bits (LSBs). However, this may not be an acceptable solution because it can decrease the dynamic range, or lead to a large accumulative error in the output.

Our new method, in contrast, can reduce the minimum bit-width required to run this fixed-point arithmetic computation code without any loss in accuracy. It can be viewed as a fully automated code transformation process that takes the original C code in Fig. 1 and user-specified ranges of the parameters as input, and returns the optimized C code in Fig. 2 as output. Our method guarantees that these two C programs are functionally equivalent and, at the same time, the one in Fig. 2 may require a smaller bit-width. In this particular example, the new code in Fig. 2 can indeed run on a 16-bit microcontroller without leading to any overflow error. Furthermore, our method ensures that the new code does not introduce additional truncation errors.

The optimization in our method is carried out by an satisfiability modulo theory (SMT) solver based inductive synthesis procedure customized specifically for efficient handling of the fixed-point arithmetic computations typically seen in embedded control software. Although recent years have seen a renewed interest in applying inductive synthesis to a wide

```
int comp(int A,int B,int H,int E,int D,int F,int K) {
 int t0,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12;
 t12 = 3 * A;
 t10 = t12 + B;
 t11 = H << 2;
 t9  = t10 + t11;
 t6  = t9 >> 3;
 t8  = 3 * E;
 t7  = t8 + D;
 t5  = t7 - 16469;
 t3  = t5 + t6;
 t4  = 12 * F;
 t2  = t3 - t4;
 t1  = t2 >> 2;
 t0  = t1 + K;
 return t0;
}
```

Fig. 1. 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.

variety of applications (see [2]–[11]), a naive application of the existing techniques would not work well in our case, due to the limited scalability and large computational overhead of the synthesis procedures. For example, our experience with Sketch [2], [12], a leading software tool in this domain, shows that for synthesizing software code that involves many fixed-point arithmetic computations, it does not scale beyond programs with 3–4 lines.

Our main contribution in this paper is proposing a new incremental inductive synthesis algorithm where the SMT solver based synthesis procedure is applied only to code regions of a bounded size, one at a time, as opposed to the entire program. We shall demonstrate in the experiments that this incremental approach is crucial for scaling up the synthesis method, ultimately allowing us to handle fixed-point arithmetic computation programs of practical size and complexity.

Our new method differs from existing methods for optimizing arithmetic computations in embedded software. Existing methods in this field, including [13] and [14], focus primarily on computing the optimal (smallest) bit-widths for all arithmetic operations and program variables in the software code. In contrast, our new method focuses on reordering the arithmetic operations and restructuring the code, which may lead to a reduction in the minimum bit-width. In other words, we are not merely computing the minimum bit-width, but also reducing it through proper code transformation. Due to the use of an SMT solver based exhaustive search, our new method can guarantee to find the best solution within the bounded design space. A more detailed discussion of related work can be found in Section IX.

We have implemented our new method in a software tool based on the Clang/LLVM compiler frontend [15] and the Yices SMT solver [16]. We evaluated the performance of this tool on a representative set of public domain software benchmarks collected from the 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 software code and, alternatively, can increase the error-free dynamic range.

To sum up, this paper makes the following contributions.

1) We propose the first SMT solver based synthesis method for incrementally optimizing fixed-point arithmetic computations in embedded C/C++ code to reduce the minimum bit-width and increase the dynamic range.
2) 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 the overall flow of our new method by using a motivating example. Then, we formally define the optimization problem and establish the notation in Section III, and present our top-level algorithm in Section V. We present our inductive synthesis procedure in Section VI. The implementation details and experimental results are given in Sections VII and VIII, respectively. We review related work in Section IX, and finally give our conclusions in Section X.

II. Motivating Example

We illustrate the overall flow of our new method using the example in Fig. 1. The program is intended to be simple for ease of presentation. In the experimental evaluation, our benchmark programs have loops and variables that are assigned more than once. Since we are mainly concerned with embedded control software, we can assume that loops are bounded and conditions are not input dependent. Therefore, loops can be handled by finite unrolling. Furthermore, in this application domain, pointers, recursive function calls, and heap allocated data structures are rarely used due to their less predictable runtime behaviors. Therefore, in the remainder of this paper, we shall focus our discussion on loop free programs with scalar variables.

Our method takes the program `comp` in Fig. 1 and a configuration file that defines the value ranges of all parameters of `comp` as input, and returns the new program in Fig. 2 as output. It starts by parsing the original C program and then constructing an abstract syntax tree (AST) for the program. Each program variable in Fig. 1 corresponds to a node in the AST. The root node is the computation result. 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 of all AST nodes. Each value range is a (min, max) pair for representing the minimum and maximum values of the node, computed using a symbolic range analysis [17].

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 detected in Fig. 1 is the subtraction at Line 13: although `t3` and `t4` can be represented in 16 bits, the subtraction may produce a value that requires more than 16 bits.

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

The region is then subjected to an inductive synthesis procedure, which will generate a functionally equivalent region that does not overflow. 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 before subtraction. 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 computation result. For Line 13, this analysis would reveal that the LSB of `t3` and the LSB of `t4` do not affect the value of the final computation output `t0`.

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 at Line 13, the next AST node that may overflow is at 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. In other words, 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 at 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 at 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 at 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 for the given input range [0, 9000]. 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 error free dynamic range than [0, 9000].

III. Preliminaries

A. Fixed-Point Notations

We follow standard practice [18] 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 $n = (N - m - 1)$. Different variables and constants in the original program are allowed to have different bit representations (different values for m), 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 $x_{N-1} \ldots x_0$, 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 ith bit. The value of α lies in the range $[-2^n, 2^n - 2^{-m}]$.

The result of fixed-point arithmetic, in general, may produce a result with as many bits as the sum of the number of bits in the two operands. Therefore, if the computation result must be fit into the same number of bits as the operands, there may be information loss. 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 – assuming that truncation is used instead of rounding. More specifically, in the event of truncation, the maximum error caused by truncation is 2^{-m}.

We define the step of a variable or a constant as the number of consecutive least significant bits (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, the step of a variable will be used to compute the truncation error margin—the LSBs whose values can be ignored. Our new 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 and constants. 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 may be represented as 10 together with the scaling factor $m = 2$, 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 $x = (c \times z) >> 3$. That is

$$\frac{c}{2^3} \times \frac{z}{2^3} = \frac{(c \times z)/2^3}{2^3}.$$

Our implementation currently supports linear fixed-point arithmetic only; therefore, we do not consider the multiplication of two variables.

It is important to note that there is no inherent difficulty in our method for handling nonlinear fixed-point arithmetic, since we encode the fixed-point computations using fixed-length bit-vector arithmetic operations in the input format of SMT solvers. Nevertheless, in the experimental evaluation, we focus on linear fixed-point arithmetic for two reasons. First, the benchmarks used in our experiments are all linear. Second, we have not evaluated the efficiency issues of modern SMT solvers related to handling nonlinear fixed-point arithmetic operations. Therefore, we leave the investigation of handling nonlinear fixed-point arithmetic for future work.

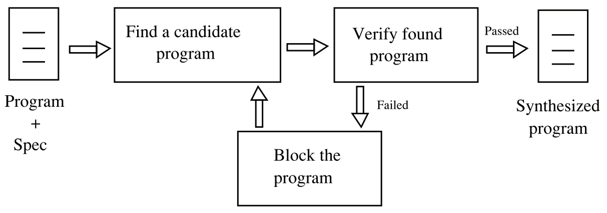


Fig. 3. Overall flow of iterative inductive synthesis procedure.

For each multiplication, i.e., $c \times z$, we also assign an accumulate flag, which can be set by the user to indicate whether the microcontroller has the capability of temporarily storing the multiplication result into two registers. Many real-world microcontrollers have been designed in this way, which effectively doubles the bit-width of the registers during multiplication. Continuing with the same example $x = (c \times z) >> 3$. If the accumulate flag is set to 1 by the user, the multiplication operation 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 their expression in the default IR representation and do not allow the user to set the accumulate flag because, to the best of our knowledge, most of the real-world microcontrollers do not have double sized registers to temporarily store the results of these operations.

C. Iterative Inductive Synthesis

We follow the iterative synthesis procedure shown in Fig. 3 while generating the optimized code region. It consists of three basic steps.

1) Given an original program or code region as input, we first compute a candidate new program that is functionally equivalent to the original program and, at the same time, is free of overflow, underflow, and truncation errors, at least for a selected set of test inputs.
2) If such candidate program exists, we try to prove that the candidate program is indeed functionally equivalent and free of implementation errors under all possible program inputs.
3) If the verification step succeeds, we are done. Otherwise, the candidate program is invalid, in which case we need to block this solution so that it will never be chosen in subsequent iterations, go back to Step 1, and try again.

In this particular application, the input program is represented by an abstract syntax tree (AST) and the set of candidate programs—or the search space—is captured by a parameterized AST. The specification is a set of constraints imposed on the two ASTs, ensuring that the candidate (parameterized) AST is functionally equivalent to the original program AST and is free of implementation errors.

The reason why we choose not to generate, in one step, a candidate program that is valid for all possible test inputs is because of performance concerns. Ultimately, the scalability of the inductive synthesis procedure is limited by the capacity of the SMT solver. A candidate program valid for all possible test inputs would be prohibitively more expensive for an SMT solver to compute than a candidate program valid for some test inputs. By separating the synthesis task into three subtasks, namely the inductive synthesis of candidate programs, the verification of candidate programs, and the iterative refinement based on the verification results, we can make all three substeps practically feasible to complete.

IV. Problem Statement

In this section, we formally define the code optimization problem.

There are two inputs to our synthesis procedure. The first input to our synthesis procedure is the source code of a C program P that takes a set of parameters and returns the result of a series of linear fixed-point arithmetic computations over these parameters. The second input to our synthesis procedure is the value ranges of all the parameters—they are called the input ranges.

Our synthesis procedure takes the aforementioned inputs and returns the C code of a new program P' such that: 1) the new program P' is functionally equivalent to the original program P—they have the same input-output relation and 2) the new program P' has potentially larger input ranges than P, or equivalently, requires a microcontroller with a smaller bit-width to execute. We do not impose constraints on the output range during the optimization process since the output of the program is completely determined by the input ranges and the C code of the program.

We focus on C programs that implement linear fixed-point arithmetic computation based control algorithms as opposed to general purpose applications. In this domain, the C programs typically do not have recursive function call or input-dependent control flow, which means that loops and function calls, if any, can be removed from the code through standard code transformation techniques such as loop unrolling and function inlining. Furthermore, since all program variables are bounded integers, the input-output relation of the program can be captured precisely by a quantifier-free first-order logic formula using bounded bit-vectors.

The grammar for the underlying arithmetic computation is defined as follows:

$$\begin{aligned} \mathit{Expression} := {} & \langle \mathit{Constant} \rangle \;\mid\; \langle \mathit{Variable} \rangle \;\mid \\ & \langle \mathit{Expression} \rangle + \langle \mathit{Expression} \rangle \;\mid \\ & \langle \mathit{Expression} \rangle - \langle \mathit{Expression} \rangle \;\mid \\ & \langle \mathit{Expression} \rangle * \langle \mathit{Constant} \rangle \;\mid \\ & \langle \mathit{Expression} \rangle \ll \langle \mathit{Constant} \rangle \;\mid \\ & \langle \mathit{Expression} \rangle \gg \langle \mathit{Constant} \rangle \end{aligned}$$

$$\mathit{Variable} := \{\text{input variables of the extracted region}\}.$$

Although this paper targets linear fixed-point arithmetic computation only, our SMT solver based synthesis method does not have inherent limitation in handling nonlinear operations. The reason is that, since we use bit-vector arithmetic to encode the fixed-point computations, we would have no problem encoding nonlinear arithmetic operations precisely and the resulting formulas would remain decidable.

We have decided to focus on linear fixed-point arithmetic for the following two reasons. First, our benchmark programs used in the experiments are all linear. Second, we have not evaluated whether modern SMT solvers are efficient enough to handle nonlinear arithmetic operations in practical settings. Therefore, we leave the optimization of nonlinear fixed-point arithmetic computations for future work.

Algorithm 1 Optimizing the Program Within its Input Range

```
1:  OPTIMIZEPROGRAM (prog, p_ranges) {
2:    ranges ← COMPUTERANGES(prog, p_ranges);
3:    ig_bits ← COMPUTEIGNOREBITS(prog);
4:    bw1 ← COMPUTEMINBITWIDTH(prog, ranges);
5:    while (true) {
6:      bw2 ← bw1 − 1;
7:      for each (Node n ∈ prog that may overflow or
          underflow) {
8:        reg ← EXTRACTREGION(prog, n);
9:        new_reg ← SYNTHESIZE(reg, bw1, bw2, ranges,
            ig_bits);
10:       if (new_reg does not exist) break;
11:       REPLACEREGION(prog, reg, new_reg);
12:     }
13:     bw1 ← bw2;
14:   }
15:   return prog;
16: }
```

V. Overall Algorithm

The elementary inductive synthesis procedure takes the existing code region as input and returns the optimized new region as output. The overall flow of our method in shown in Algorithm 1. The input to our procedure OPTIMIZEPROGRAM includes the original program and the value ranges of all the parameters. First, we invoke COMPUTERANGES to compute the value ranges of all nonleaf AST nodes. Then, we invoke COMPUTEIGNOREBITS to compute the truncation error margins for all AST nodes. The truncation error margin of an AST node is the LSBs whose values can be ignored. We also compute the bit-width ($bw1$) required by the original program to run within the given input range.

After the bit-width of the original program ($bw1$) is determined, we enter the while-loop (Lines 5-14) to iteratively optimize the program. In each iteration, we try to reduce the minimum bit-width from $bw1$ to $bw2$. The while-loop terminates as soon as a call to the inductive synthesis procedure fails to return the new region.

Within each while-loop iteration, we first search for nodes that may overflow or underflow when the new bit-width ($bw2$) is used instead of the old bit-width ($bw1$). We traverse these nodes in a breadth-first search (BFS) order, i.e., from the return value of the program to the input variables. For each AST node, we invoke EXTRACTREGION to extract a neighboring region for optimization and then invoke the elementary inductive synthesis procedure (Line 9). If successful, the inductive synthesis procedure would return a new code region, which is functionally equivalent to the extracted region but free of overflow and underflow errors. It also ensures that the new region does 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 for optimization 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 during previous iterations. 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 better 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 existing inductive synthesis tools, such as the reference implementations of the syntax guided synthesis (SyGuS [10]) procedure, to the entire program, provided that the fixed-point arithmetic optimization problem is modeled in their input language. In practice, however, such a monolithic optimization approach does not work. Indeed, our experience with the Sketch tool [2], [12] shows that it cannot scale beyond parameterized fixed-point arithmetic computation code of 2-3 lines. Although we expect SMT solvers and heuristic search techniques to continue improving in the coming years, it is unlikely that a monolithic inductive synthesis procedure will scale up to large programs—this is consistent with what other researchers in the field have observed [10], [19].

Therefore, we have implemented our customized inductive synthesis procedure, which is optimized to handle fixed-point arithmetic computations efficiently. In addition, we bound the size of the extracted region so that the elementary inductive synthesis procedure is applied only to these small code regions individually in the context of incremental optimization. In practice, we have found that the extracted code region needs to be bounded to an AST with at most five node levels to allow the SMT solver to complete in a reasonable amount of time – it means that the code region can represent up to 63 AST nodes. Furthermore, we have found that bounding the search to AST regions with 3 or 4 levels often can produce as decent results.

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. Then, we compute the step for all other AST nodes in a bottom-up topological order. 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.

1) $\mathsf{step}(x * y) = \mathsf{step}(x) + \mathsf{step}(y)$;
2) $\mathsf{step}(x + y) = \min(\mathsf{step}(x), \mathsf{step}(y))$;
3) $\mathsf{step}(x - y) = \min(\mathsf{step}(x), \mathsf{step}(y))$;
4) $\mathsf{step}(x << c) = \mathsf{step}(x) + c$;
5) $\mathsf{step}(x >> c) = \max(\mathsf{step}(x) - c, 0)$.

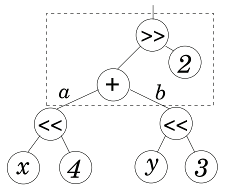


Fig. 4. Extracted region.

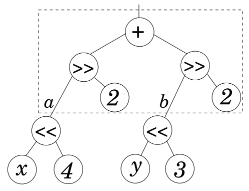


Fig. 5. Synthesized region.

The ignore bits are the 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 synthesize more compact new regions.

To further clarify this, consider the example in Fig. 4, where the extracted region for optimization 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 three lowest significant bits to ignore. In other words, we have the freedom to truncate up to three 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 code region as shown in Fig. 5.

Notice that, even if we do not consider the ignore bits of a and b, our method can still synthesize a new region same as the one in Fig. 5 to remove the overflowing node in the above example. However, in such case, the extracted region would have to be larger. At the very least, the extracted region would need to include all the AST nodes in Fig. 4. The synthesized new region would include all the nine AST nodes in Fig. 5. However, this would also be more expensive computationally, leading to significantly longer execution time for the synthesis procedure.

VI. 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 functionally equivalent to the extracted region, and at the same time, free of implementation errors, at least for the selected set of test cases and 2) check if the new code region is a valid solution in the full input range. If the new code region is not

Algorithm 2 Inductively Synthesizing the New Code Region

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

valid under a certain input condition, then block this candidate region (bad solution) and try again.

Algorithm 2 shows the pseudocode of our synthesis subprocedure, which computes a new region (*new_r*) of bit-width *bw*2 such that it is equivalent to the original region (*reg*) of bit-width *bw*1, under the value ranges specified in *ranges* while considering the truncation error margins specified in *ig_bits*.

The procedure starts by initializing *blockedRegs* and *testSet* to empty sets, where *testSet* consists of the test cases used for inductively generating (guessing) a new region, and *blockedRegs* consists of the previously explored regions that fail the equivalence check (bad solutions).

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. If size exceeds a predetermined bound, it means that we have proved that no solution exists in this bounded 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 (bad solutions) in *blockedRegs*. Subroutine COMPDIFF formally verifies the equivalence of the extracted region (*reg*) and the new region (*new_r*), and returns a concrete test case if they are not equivalent. The concrete test case would be one that can force the two regions to behave differently.

A. Constructing the New Region Skeleton

First, we generate a skeleton of the new region, which is a parameterized AST capable of representing any linear fixed-point 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. 6 shows an example skeleton of seven AST nodes. Here, *Op* represents any binary arithmetic operator and $V|C$ represents a leaf node, which is either a variable or a constant.

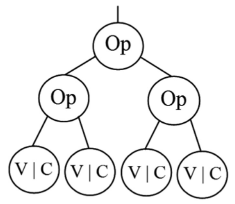


Fig. 6. Skeleton of seven AST nodes.

For each AST node in the skeleton, we assign an auxiliary variable called the selector, whose value determines the node type. For example, a parameterized 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 parameterized 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 compute a set of suitable values for all these selector variables.

B. Inductively Generating the New Region

To generate the new region inductively, we need a representative set of test cases for the extracted region. These are test values for the input variables of the region, and ideally, should include both the corner cases and the intermediate values. Since the arithmetic computations that we are concerned with 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 based on both the original region and the parameterized AST.

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

$$\Phi = \Phi_{\text{reg}} \wedge \Phi_{\text{skel}} \wedge \Phi_{\text{sameI}} \wedge \Phi_{\text{sameO}} \wedge \Phi_{\text{tests}} \wedge \Phi_{\text{blocked}}$$

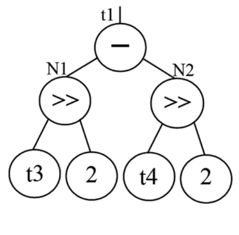


Fig. 7. Synthesized new region.

where the subformulas are defined as follows.

1) *Extracted Region (Φ_{reg}):* It encodes the input-output relation of the extracted region by using bit-vector arithmetic, where the bit-width of the AST is *bw1*.
2) *New Region Skeleton (Φ_{skel}):* It encodes the input-output relation of the skeleton by using bit-vector arithmetic, where the bit-width is *bw2*.
3) *Same Input Values (Φ_{sameI}):* It asserts that the input variables of the two regions must have the same values.
4) *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.
5) *Test Cases (Φ_{tests}):* It asserts that the input variables must adopt concrete values from the given test cases.
6) *Blocked Solutions (Φ_{blocked}):* It asserts that the selector variables should not take values that represent any previously explored (bad) solution.

If the formula Φ is proved by the SMT solver to be unsatisfiable, it means that 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 for optimization in Section II, whose skeleton is shown in Fig. 6. The new region generated from the skeleton in Fig. 6 is shown in Fig. 7. The new region is constructed by setting the selector variables of all AST nodes the values in a satisfying assignment returned by the SMT solver.

C. Checking the Validity of the New Region

The candidate new region is guaranteed to be equivalent to the extracted region over the selected 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 if and only if the two regions are not equivalent; that is, Ψ is satisfiable if and only if there exists a test case that can differentiate them. Formula Ψ is defined as follows:

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

where the subformulas are defined as follows.

1) *Extracted Region (Φ_{reg}):* It encodes the input-output relation of the extracted region by using bit-vector arithmetic, where the bit-width of the AST is *bw1*.
2) *New Region ($\Phi_{\text{new\_reg}}$):* It encodes the input-output relation of the candidate new region in bit-vector arithmetic, where the bit-width of the AST is *bw2*.
3) *Different Output Values (Φ_{diffO}):* It asserts that the output variables of the two regions have different values.
4) *Value Ranges (Φ_{ranges}):* It asserts that all input variables should stay within their precomputed value ranges, since we are not interested in checking the equivalence of the two regions outside these designated value ranges.
5) *Ignore Bits ($\Phi_{\text{ig\_bits}}$):* 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 the formula 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 valid. In this case, we add this bad solution to *blockedRegs* and try again. The blocking of an invalid solution follows the counter-example guided inductive synthesis algorithm [12], [20], where the blocked solutions are encoded as additional constraints 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 bad solution will not be returned.

VII. 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 [15] and the Yices SMT solver [16]. 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 for optimization.

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 existing inductive synthesis tools such as Sketch [12], [20], provided that our new region skeleton is 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 enough: synthesizing a new region with a size bound of more than 2 would cause Sketch to 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 with a large bit-width. Second, inductive synthesis, in general, may not be suitable for handling arbitrarily large programs of fixed-point arithmetic computations.

Due to the aforementioned scalability problem encountered by using Sketch, we have implemented our own inductive synthesis procedure using the Yices SMT solver and optimize it for efficient handling of fixed-point arithmetic operations, e.g., by designing SMT encoding schemes for exploiting the AST structures encountered in this application. Our experimental evaluation shows that the new procedure is significantly more efficient: instead of running out of memory for a parametrized AST with a size bound of 2 or 3, it now can handle the skeleton with a size bound of up to 5 (representing up to 63 AST nodes). Nevertheless, this improvement alone is not sufficient for supporting the whole-program optimization.

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

Therefore, we have implemented the new incremental optimization method, which applies the SMT solver based 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 now routinely handle programs of practical size and complexity.

VIII. EVALUATION

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

1) How much can our method reduce the minimum bit-width required for the program to run in the given input range without additional implementation errors?
2) How much can our method increase the dynamic range of the program for the given bit-width?
3) If both the original program and the optimized program are forced to run with a certain 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 digital signal processing (DSP) applications. They come from various sources including papers, textbooks, and the output of automated code generation tools such as the Real Time Workshop of the MATLAB toolkit. 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 type of cyber-physical system (CPS) software targeted by our new method, these are representative programs with realistic size and complexity.

Table I shows the statistics of the benchmarks. The first example, taken from [21], is a 3×3 Sobel digital filter that is widely used in image processing applications. The second example, taken from [14], is a bicycle controller optimally synthesized for a custom-designed microprocessor with double-sized internal registers during multiplication. The third example is a locomotive controller generated by using Fixed Point Advisor and Real Time Workshop of the MATLAB toolkit [22]. The fourth example, taken from [23], is an inverse discrete cosine transform (IDCT), which is widely used in mobile communication and image compression applications. The fifth example is the fixed-point version of a control rule implementation from [22]. The sixth example is a 5×5 kernel sized difference image filter [24]. The seventh example is a fast Fourier transform (FFT) implementation, where the floating-point version was taken from [25] and then converted to a fixed-point program by changing all `double` variables into `int` variables without modifying or reordering any of its instructions. The eighth example is the inverse fast Fourier transform (IFFT) for the FFT implemented in the seventh example. None of the benchmarks was modified from their original forms in any way to give performance advantage to our method.

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 OVERFLOW/UNDERFLOW FREE OUTPUT RANGE

benchmark	bit	original	optimized	%
Sobel Image	32	[0, 16320]	[-49184, 65504]	602
Bicycle	32	[-5.3*10^8, 5.3*1^8]	[-1.5*10^9, 1.5*10^9]	194
Locomotive	64	[-3.6*10^{18}, 5.0*10^{18}]	[-3.9*10^{18}, 5.2*10^{18}]	5
IDCT	32	[-1.4*10^6, 2.9*10^8]	[-1.9*10^7, 3.9*10^8]	40
Controller	32	Out1 [0, 1.0*10^9]	Out1 [0, 1.4*10^9]	32
		Out2 [0, 1.0*10^9]	Out2 [0, 1.4*10^9]	31
		Out3 [0, 1.0*10^9]	Out3 [0, 1.4*10^9]	31
Diff. Image	32	[0, 1.3*10^8]	[-1.0*10^9, 1.1*10^9]	1515
FFT (N=8)	32	[25600, 25600]	[25600, 25600]	0
IFFT (N=8)	32	[-1.3*10^8, 2.6*10^8]	[-2.6*10^8, 5.3*10^8]	103

All experiments were conducted on a machine with a 3.4 GHz Intel i7-2600 CPU, 3.3 GB 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. Tables II and III show the experimental results. In both tables, 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 by code transformation. Together, they represent 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 and additional truncation errors for

TABLE IV
INCREASE IN THE MINIMUM AND AVERAGE BIT-WIDTHS

Name of Benchmark	Original (bit-width)		Optimized (bit-width)	
	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 V
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 VI
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

TABLE VII
SYSTEMATIC BREAKDOWN OF THE OPTIMIZATION PROCESS

Name of the Benchmark	Synth. Time	Verif. Time	Tests	Iterations
Sobel Image filter (3x3)	69%	31%	6.1	1.7
Bicycle controller	64%	36%	8.5	1.0
Locomotive controller	72%	28%	9.0	2.0
IDCT (N=8)	71%	28%	5.0	2.0
Control. Impl.	58%	42%	6.3	6.0
Diff. image filter (5x5)	72%	28%	6.4	1.8
FFT (N=8)	73%	27%	17.0	2.0
IFFT (N=8)	75%	25%	5.0	3.0

TABLE VIII
QUANTIFYING THE IMPACT OF THE INITIAL SET OF TEST CASES

Name of the Benchmark	Speedup (using the initial set versus empty set)
Sobel Image filter (3x3)	0.9
Bicycle controller	1.7
Locomotive controller	7.8
IDCT (N=8)	2.0
Control. Impl.	1.1
Diff. image filter (5x5)	0.9
FFT (N=8)	1.0
IFFT (N=8)	2.1

the given input range. The experimental results are shown in Table IV. Column 1 is the name of the benchmark. Column 2 is the minimum bit-width of the original program to avoid overflow, underflow, and truncation errors 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, underflow, and truncation errors and Column 5 is the average bit-width for all program variables.

Our results show that overall 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. In other words, with 17 bits, the program would have the same precision and error-free dynamic range as with 32 bits. After our SMT solver based optimization, the minimum bit-width is reduced to 15 bits. This is practically 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 V 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.

C. Statistics

We also show, in Table VI, 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, the execution time is no longer exponentially dependent of the whole-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 than for the other examples because it has a larger original bit-width (64-bit) – the other examples are all 32-bit.

To understanding the computational cost of our optimization process as shown in Table VI, we provide a systematic breakdown of the total time into time required to perform different steps of the algorithm, the average number of test cases used to generate a new region, and the average number of inductive iterations required. The results are shown in Table VII.

To evaluate the impact of the heuristically selected initial set of test cases, we compared the runtime performance of our synthesis procedure with and without the initial set. Recall that in Section VI-B, we have described how to choose the initial set of test cases for generating the new code region inductively. The main idea is to choose both representative and corner values so that the synthesis procedure can start with a good set of examples. Table VIII shows the speedup of the synthesis procedure using the initial set versus using an empty set. The results show that using the heuristically selected test cases to start the synthesis procedure is advantageous in general, although for two examples, there is a slight slowdown.

IX. Related Work

This paper is an extended version of our recent work [1] that proposed the incremental SMT solver based inductive synthesis method for optimizing embedded software code for the first time. In this extended version, we have provided a more detailed description of the method and algorithms and presented more experimental results.

Our new method incrementally transforms 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 [2], [12]. However, our method is significantly different in that it has an implementation designed for handling linear fixed-point arithmetic computations more efficiently. Furthermore, we apply inductive synthesis incrementally to code regions of a bounded size, one at a time, as opposed to the entire program.

Gulwani *et al.* [6] 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 min. Jha *et al.* [4] use the same symbolic encoding as in [6] but replace the logical specification of the desired program by an input-output oracle.

The SCIDUCTION tool implemented by Jha [13] 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 [14], 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 does not attempt to change the structure of the original code or synthesize completely new code in order to reduce the bit-width.

Darulova *et al.* [26] proposed a method for compiling real-valued arithmetic expressions to fixed-point arithmetic programs to minimize the discrepancy between the fixed-point values and the real values. Their method uses genetic programming, which mutates the order of the original arithmetic expressions to find better fixed-point representations. The method differs from ours in three aspects. First, their method takes a real-valued expression in MATLAB format as input and returns a fixed-point arithmetic program as output whereas our method transforms an existing fixed-point C program into another fixed-point C program—this also makes experimental comparison of the two approaches difficult to conduct. Second, their method relies on genetic programming, which consists of random mutation and filtering of the mutants, whereas our methods relies on exhaustive search via an SMT solver. Third, their method does not employ incremental inductive analysis, which is one of the main contributions of our work.

Our new method is also related to the various superoptimization techniques that are becoming popular in compilers in recent years [27]–[29]. Superoptimizers are more powerful than conventional compiler based optimizations that rely on matching known code patterns and then applying predetermined transformation rules. In contrast, superoptimizers often perform a more involved search in the implementation space of a set of valid instruction sequences, for example, to optimize performance-critical inner loops. However, to the best of our knowledge, there has not been any existing superoptimizer that can be used to increase the error free dynamic range, or to minimize the minimum bit-width, of fixed-point arithmetic computations in embedded C programs.

X. Conclusion

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 error-free 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.

References

[1] H. Eldib and C. Wang, "An SMT based method for optimizing arithmetic computations in embedded software code," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design*, Portland, OR, USA, Oct. 2013, pp. 129–136.

[2] A. Solar-Lezama, C. G. Jones, and R. Bodík, "Sketching concurrent data structures," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2008, pp. 136–148.

[3] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2008, pp. 281–292.

[4] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proc. 32nd ACM/IEEE Int. Conf. Software Eng. (ICSE)*, May 2010, pp. 215–224.

[5] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proc. ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, Austin, TX, USA, 2011, pp. 317–330.

[6] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2011, pp. 62–73.

[7] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2011, pp. 317–328.

[8] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed completion of partial expressions," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, Beijing, China, 2012, pp. 275–286.

[9] R. Singh and S. Gulwani, "Synthesizing number transformations from input-output examples," in *Proc. Int. Conf. Comput. Aided Verification*, Berkeley, CA, USA, Jul. 2012, pp. 634–651.

[10] R. Alur *et al.*, "Syntax-guided synthesis," in *Proc. Int. Conf. Formal Methods Comput.-Aided Design*, 2013, pp. 1–17.

[11] H. Eldib and C. Wang, "Synthesis of masking countermeasures against side channel attacks," in *Proc. Int. Conf. Comput. Aided Verification*, Vienna, Austria, Jul. 2014, pp. 114–130.

[12] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu, “Programming by sketching for bit-streaming programs,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2005, pp. 281–294.

[13] S. K. Jha, “Towards automated system synthesis using sciduction,” Ph.D. dissertation, EECS Dept., Univ. California, Berkeley, CA, USA, Nov. 2011.

[14] R. Majumdar, I. Saha, and M. Zamani, “Synthesis of minimal-error control software,” in *Proc. ACM Int. Conf. Embedded Softw.*, 2012, pp. 123–132.

[15] C. Lattner and V. Adve, “The LLVM instruction set and compilation strategy,” CS Dept., Univ. Illinois, Urbana-Champaign, IL, USA, Tech. Rep. UIUCDCS-R-2002-2292, Aug. 2002.

[16] B. Dutertre and L. de Moura, “A fast linear-arithmetic solver for DPLL(T),” in *Proc. Int. Conf. Comput. Aided Verification*, Seattle, WA, USA, Aug. 2006, pp. 81–94.

[17] R. Rugina and M. C. Rinard, “Symbolic bounds analysis of pointers, array indices, and accessed memory regions,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2000, pp. 182–195.

[18] R. Yates. (2013, Jan. 2). *Fixed-point arithmetic: An introduction*. Digital Signal Labs, Technical Reference [Online]. Available: http://www.digitalsignallabs.com/fp.pdf

[19] T. Akiba *et al.*, “Calibrating research in program synthesis using 72,000 hours of programmer time,” MSR, Redmond, WA, USA, Tech. Rep., 2013 [Online]. Available: http://research.microsoft.com/enus/um/people/nswamy/papers/calibrating-program-synthesis.pdf

[20] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat, “Combinatorial sketching for finite programs,” in *Proc. Archit. Support Program. Lang. Operat. Syst.*, San Jose, CA, USA, Dec. 2006, pp. 404–415.

[21] S. Qureshi, *Embedded Image Processing on the TMS320C6000 DSP*. New York, NY, USA: Springer, 2005.

[22] A. Martinez, R. Majumdar, I. Saha, and P. Tabuada, “Automatic verification of control system implementations,” in *Proc. ACM Int. Conf. Embedded Softw.*, Scottsdale, AZ, USA, 2010, pp. 9–18.

[23] S. Kim, K.-I. Kum, and W. Sung, “Fixed-point optimization utility for C and C++ based digital signal processing programs,” *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 45, no. 11, pp. 1455–1464, Nov. 1998.

[24] W. Burger and M. Burge, *Digital Image Processing*. New York, NY, USA: Springer, 2008.

[25] J. Xiong, J. R. Johnson, R. W. Johnson, and D. A. Padua, “SPL: A language and compiler for DSP algorithms,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2001, pp. 298–308.

[26] E. Darulova, V. Kuncak, R. Majumdar, and I. Saha, “Synthesis of fixed-point programs,” in *Proc. ACM Int. Conf. Embedded Softw.*, Montreal, QC, Canada, 2013, pp. 1–10.

[27] R. Joshi, G. Nelson, and K. H. Randall, “Denali: A goal-directed superoptimizer,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implementation*, 2002, pp. 304–314.

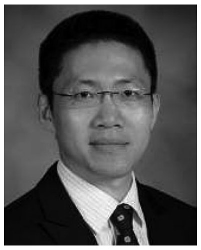
[28] S. Bansal and A. Aiken, “Automatic generation of peephole superoptimizers,” in *Proc. Archit. Support Program. Lang. Operat. Syst.*, 2006, pp. 394–403.

[29] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic superoptimization,” in *Proc. Archit. Support Program. Lang. Operat. Syst.*, Houston, TX, USA, 2013, pp. 305–316.

Hassan Eldib received the B.S. and M.S. degrees from the Arab Academy for Science and Technology and Maritime Transport, Alexandria, Egypt, in 2006 and 2009, respectively. He is currently pursuing the Ph.D. degree from the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA.

His current research interests include developing automated synthesis and verification methods for embedded control software and cryptographic software.

Mr. Eldib was a recipient of the 2013 FMCAD Best Paper Award.



Chao Wang (M’02) received the Ph.D. degree from the University of Colorado, Boulder, CO, USA, in 2004.

He is currently an Assistant Professor with the Bradley Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, USA. He has published a book and over 50 papers in top venues in the areas of formal verification, software verification, program analysis, and program synthesis.

Dr. Wang received the FMCAD Best Paper Award in 2013, the ACM SIGSOFT Distinguished Paper Award in 2010, the ACM TODAES Best Journal Paper of the Year Award in 2008, and the ACM SIGDA Outstanding Ph.D. Dissertation Award in 2004. He was a recipient of the NSF Faculty CAREER Award in 2012 and the ONR Young Investigator Award in 2013.