Synthesis of Optimal Fixed-Point Implementation of Numerical Software Routines

Susmit Jha[1,2] and Sanjit A. Seshia[2]

[1] Strategic CAD Labs, Intel
susmit.jha@intel.com
[2] UC Berkeley
jha, sseshia@eecs.berkeley.edu

Abstract. In this paper, we present an automated technique `swati`: *Synthesizing Wordlengths Automatically Using Testing and Induction*, which uses a combination of Nelder-Mead optimization based testing, and induction from examples to automatically synthesize optimal fixedpoint implementation of numerical routines. The design of numerical software is commonly done using floating-point arithmetic in design-environments such as Matlab. However, these designs are often implemented using fixed-point arithmetic for speed and efficiency reasons especially in embedded systems. The fixed-point implementation reduces implementation cost, provides better performance, and reduces power consumption. The conversion from floating-point designs to fixed-point code is subject to two opposing constraints: (i) the word-width of fixed-point types must be minimized, and (ii) the outputs of the fixed-point program must be accurate. In this paper, we propose a new solution to this problem. Our technique takes the floating-point program, specified accuracy and an implementation cost model and provides the fixed-point program with specified accuracy and optimal implementation cost. We demonstrate the effectiveness of our approach on a set of examples from the domain of automated control, robotics and digital signal processing.

1 Introduction

Numerical software forms a critical component of systems in domains such as robotics, automated control and digital signal processing. These numerical routines have two important characteristics. First, these routines are procedures that compute some mathematical functions designed ignoring precision issues of fixed-point arithmetic. Design environments such as Simulink/Stateflow and LabVIEW allow design and simulation of numerical routines using floating-point arithmetic that closely resembles the more intuitive real arithmetic. Second, the implementation of these numerical routines run in resource-constrained environments, requiring their optimization for low resource cost and high performance. It is common for embedded platforms to have processors without floating-point units due to their added cost and performance penalty. The signal processing/control engineer must thus redesign her floating-point program to instead use *fixed-point arithmetic*. Each floating-point variable and operation in the original program is simply replaced by a corresponding fixed-point variable and operation, so the basic structure of the program does not change. The tricky part of the redesign process is to find the *optimal fixed-point types*, viz., the optimal wordlengths (bit-widths) of fixed-point variables, so that the implementation on the platform is optimal — lowest cost and highest performance — *and* the resulting fixed-point program is sufficiently

accurate. The following novel contributions are made in this paper to address this problem:

- We present a new approach for inductive synthesis of fixed-point programs from floating-point versions. The novelty stems in part from our use of optimization: we not only use optimization routines to minimize fixed-point types (bit-widths of fixed-point variables), as previous approaches have, but also show how to use an optimization oracle to systematically test the program and generate input-output examples for inductive synthesis.
- We illustrate the practical effectiveness of our technique on programs drawn from the domains of digital signal processing and control theory. For the control theory examples, we not only exhibit the synthesized fixed-point programs, but also show that these programs, when integrated in a feedback loop with the rest of the system, perform as accurately as the original floating-point versions.

2 Preliminaries

Floating-point arithmetic [8] is a system for approximately representing real numbers that supports a wide range of values. It approximates a real number using a fixed number of significant digits scaled using an exponent. The floating-point system is so called because the radix point can *float* anywhere relative to the significant digits of the number. This is in contrast to fixed-point arithmetic [23] in which there are a fixed number of digits and the radix point is also fixed. Due to this feature, a floating-point representation can represent a much wider range of values with the same number of digits. The most common floating-point representation used in computers is that defined by the IEEE 754 Standard [1]. In spite of the benefits of floating-point arithmetic, embedded systems often use fixed-point arithmetic to reduce resource cost and improve performance. A fixed-point number consists of a sign mode bit, an integer part and a fractional part. We denote the fixed-point type of a variable x by $\mathbf{fx}\tau(x)$. Formally, a fixed-point type is a triple:

$$\langle \texttt{Signedness}, \texttt{IWL}, \texttt{FWL} \rangle.$$

The sign mode bit `Signedness` is 0 if the data is unsigned and is 1 if the data is signed. The length of the integer part is called the integer wordlength (`IWL`) and the length of the fractional part is called the fractional wordlength (`FWL`). The fixed-point wordlength (`WL`) is the sum of the integer wordlength and fractional wordlength; that is, $\texttt{WL} = \texttt{IWL} + \texttt{FWL}$. The operations supported by fixed-point arithmetic are the same as those in the floating-point arithmetic standard [1] but the semantics can differ on custom hardware. For example, the rounding mode for arithmetic operations could be different, and the result could be specified to saturate or overflow/underflow in case the wordlength of a variable is not sufficient to store a computed result. One complete semantics of fixed-point operation is provided with the Fixed-point Toolbox in Matlab [2]. The range of the fixed-point number is much smaller compared to the range of floating-point numbers for the same number of bits since the radix point is fixed and no dynamic adjustment of precision is possible. Translating a floating-point program into fixed-point program is non-trivial and requires careful consideration of loss of precision and range. The integer wordlengths and fractional wordlengths of the fixed-point variables need to be carefully selected to ensure that the computation remains accurate to a specified threshold. Please

refer to the extended version of the paper available as technical report [12] for more background discussion.

3 Problem Definition

We introduce a simple illustrative example to explain the problem of synthesizing an optimal fixed-point program from a floating-point program, and then present the formal problem definition.

Floating-point Implementation: Given a floating-point program, we need to synthesize fixed-point type for each floating-point variable.

Example 1: The floating-point program in this example 1 takes `radius` as the input, and computes the corresponding `area` of the circle. Notice that the fixed-point program is essentially identical to the floating-point version, except that the fixed-point types of variables $\texttt{mypi}, \texttt{radius}, \texttt{t}$ and `area` must be identified. Recall that the fixed-point type is a triple $\langle \mathtt{s_j}, \mathtt{iwl_j}, \mathtt{fwl_j} \rangle$ for j-th variable where $\mathtt{s_j}$ denotes the `Signedness` of the variable, $\mathtt{iwl_j}$ denotes the integer wordlength and $\mathtt{fwl_j}$ denotes the fraction wordlength.

Procedure 1 Floating-point program to compute circle area

Input: `radius`
Output: `area`
double $\texttt{mypi}, \texttt{radius}, \texttt{t}, \texttt{area}$
$\texttt{mypi} = 3.14159265358979323846$
$\texttt{t} = \texttt{radius} \times \texttt{radius}$
$\texttt{area} = \texttt{mypi} \times \texttt{t}$
return `area`

Procedure 2 Fixed-point program to compute circle area

Input: `radius`, $\langle \mathtt{s_j}, \mathtt{iwl_j}, \mathtt{fwl_j} \rangle$ for $\mathtt{j} = 1, 2, 3, 4$
Output: `area`
$\mathbf{fx}\langle \mathbf{s_1}, \mathbf{iwl_1}, \mathbf{fwl_1} \rangle$ `mypi`
$\mathbf{fx}\langle \mathbf{s_2}, \mathbf{iwl_2}, \mathbf{fwl_2} \rangle$ `radius`
$\mathbf{fx}\langle \mathbf{s_3}, \mathbf{iwl_3}, \mathbf{fwl_3} \rangle$ `t`
$\mathbf{fx}\langle \mathbf{s_4}, \mathbf{iwl_4}, \mathbf{fwl_4} \rangle$ `area`
$\texttt{mypi} = 3.14159265358979323846$
$\texttt{t} = \texttt{radius} \times \texttt{radius}$
$\texttt{area} = \texttt{mypi} \times \texttt{t}$
return `area`

We use $F_{fl}(X)$ to denote the floating-point program with inputs $X = \langle x_1, x_2, \ldots, x_n \rangle$. $F_{fx}(X, \mathbf{fx}\tau)$ denotes the fixed-point version of the program, where the fixed-point type of a variable $x \in X$ is $\mathbf{fx}\tau(x)$. Note that the fixed-point types in $F_{fx}(X, \mathbf{fx}\tau)$ are defined by the mapping $\mathbf{fx}\tau$.

Input Domain: The context in which a fixed-point program $F_{fx}(X, \mathbf{fx}\tau)$ is executed often provides a precondition that must be satisfied by valid inputs $\langle x_1, x_2, \ldots, x_n \rangle$. This defines the input domain denoted by $Dom(X)$.

Example 2: In our example of computing the area of a circle, suppose that we are only interested in the radii in the range $[0.1, 2.0)$. Then, the input domain $Dom(\texttt{radius})$ is

$$\texttt{radius} \geq 0.1 \wedge \texttt{radius} < 2.0$$

Correctness Condition for Accuracy: The correctness condition specifies an error function $Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau))$, and a maximum error threshold `maxError`. The error function and error threshold together define a bound on the "distance" between outputs generated by the floating-point and fixed-point programs respectively. An *accurate* fixed-point program is one whose error function lies within the error threshold for all inputs in the input domain. Some common error functions are:

- Absolute difference between the floating-point function and fixed-point function: $|F_{fl}(X) - F_{fx}(X, \mathbf{fx}\tau)|$
- Relative difference between the floating-point function and fixed-point function: $\left|\frac{F_{fl}(X) - F_{fx}(X, \mathbf{fx}\tau)}{F_{fl}(X)}\right|$
- Moderated relative difference: $\left|\frac{F_{fl}(X) - F_{fx}(X, \mathbf{fx}\tau)}{F_{fl}(X) + \delta}\right|$. This approaches the relative difference for $F_{fl}(X) >> \delta$ and approaches a weighted absolute difference for $F_{fl}(X) << \delta$. When $F_{fl}(X)$ can be zero for some values of X, the moderated relative difference remains bounded unlike the relative difference which becomes unbounded.

The *correctness condition for accuracy* requires that for all inputs in the provided *input domain* $Dom(X)$, the error function $Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau))$ is below the specified threshold `maxError`; i.e.,

$$\forall X \in Dom(X)\ .\ Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau)) \leq \texttt{maxError}$$

Example 3: In our running example of computing the area of a circle, the error function is chosen to be relative difference, the error threshold 0.01, and thus the correctness condition is $\forall \texttt{radius},\ s.t.\ \texttt{radius} \geq 0.1 \wedge \texttt{radius} < 2.0$

$$\left|\frac{F_{fl}(\texttt{radius}) - F_{fx}(\texttt{radius}, \mathbf{fx}\tau)}{F_{fl}(\texttt{radius})}\right| \leq 0.01$$

Implementation Cost Model: The *cost model* of the fixed-point program is a function mapping fixed-point types to a real number. For a given fixed-point program $F_{fx}(X, \mathbf{fx}\tau)$, let $X = \{t_1, t_2, \ldots, t_k\}$ be the set of fixed-point program variables with corresponding types $\{\mathbf{fx}\tau(t_1), \mathbf{fx}\tau(t_2), \ldots, \mathbf{fx}\tau(t_k)\}$. Then the cost model (or simply *cost*) of F_{fx} is a function

$$\texttt{cost} : (\mathbf{fx}\tau(t_1), \mathbf{fx}\tau(t_2), \ldots, \mathbf{fx}\tau(t_k)) \rightarrow \mathbb{R}$$

In practice, `cost` is often just a function of the total wordlengths ($\texttt{WL} = \texttt{IWL} + \texttt{FWL}$) of the variables. It can incorporate hardware implementation metrics such as area, power and delay. A number of cost models are available in the literature [15, 16, 5, 6], and all of these can be used in our approach.

Example 4: The cost model proposed by Constantinides et al [6] for the running example yields the following cost function. We use this cost model in all our examples.

$$\texttt{cost}(\mathbf{fx}\tau(\texttt{mypi}), \mathbf{fx}\tau(\texttt{radius}), \mathbf{fx}\tau(\texttt{t}), \mathbf{fx}\tau(\texttt{area})) =$$

$$\texttt{cdelay}(\texttt{WL}(\texttt{mypi})) + \texttt{cmul}(\texttt{WL}(\texttt{radius}), \texttt{WL}(\texttt{radius}), \texttt{WL}(\texttt{t}))$$

$$+\texttt{cmul}(\texttt{WL}(\texttt{mypi}), \texttt{WL}(\texttt{t}), \texttt{WL}(\texttt{area}))\ , \texttt{where}$$

$$\texttt{cdelay}(l) = l + 1 \texttt{ and } \texttt{cmul}(l_1, l_2, l) = 0.6 \times (l_1 + 1) * l_2 - 0.85 * (l_1 + l_2 - l)$$

The area of a multiplier grows almost linearly with both the coefficients and the data wordlength. The first term in the Constantinides model represents this cost. The second term represents the area cost of computational elements required only for carry propagation. The coefficients 0.6 and 0.85 were obtained through least-squared fitting to area of several hundred multipliers of different coefficient value and width [6].

Problem Definition

Definition 1 (*Optimal Fixed-point Types Synthesis).* *The optimal fixed-point types synthesis problem is as follows. Given a floating-point program $F_{fx}(X, \mathbf{fx}\tau(T))$ with variables T, an input domain $Dom(X)$, a correctness condition $Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau(T))) \leq \texttt{maxError}$, and a cost model $\texttt{cost}(\mathbf{fx}\tau(t_1), \mathbf{fx}\tau(t_2), \ldots, \mathbf{fx}\tau(t_k))$, the optimal fixed-point types synthesis problem is to discover fixed-point types*

$$\mathbf{fx}\tau^*(T) = \{\mathbf{fx}\tau^*(t_1), \mathbf{fx}\tau^*(t_2), \ldots, \mathbf{fx}\tau^*(t_k)\}$$

such that the fixed-point program $F_{fl}(X)$ with the above types for fixed-point variables satisfies the correctness condition for accuracy, that is,

$$(a)\ \forall X \in Dom(X)\ .\ Err(F_{fl}(X), F_{fx}(X, \mathbf{fx}\tau^*(T))) \leq \texttt{maxError}$$

and has minimal cost with respect to the given cost function among all fixed-point types that satisfy condition (a), that is,

$$(b)\ \mathbf{fx}\tau^* = \underset{\mathbf{fx}\tau \text{ satisfies (a)}}{\operatorname{argmin}} \texttt{cost}(\mathbf{fx}\tau(T))$$

Our goal is to automate this search for optimal fixed-point types. We illustrate this problem using the running example below.

Example 5: In our running example of computing the *area of a circle*, we need to discover $\mathbf{fx}\tau^*(\texttt{mypi})$, $\mathbf{fx}\tau^*(\texttt{radius})$, $\mathbf{fx}\tau^*(\texttt{t})$ and $\mathbf{fx}\tau^*(\texttt{area})$ such that

(a) the fixed-point program with the given fixed-point types satisfies the correctness condition; that is, $\forall \texttt{radius},\ s.t.,\ \texttt{radius} \geq 0.1 \wedge \texttt{radius} < 2.0$

$$\left|\frac{F_{fl}(\texttt{radius}) - F_{fx}(\texttt{radius}, \mathbf{fx}\tau^*)}{F_{fl}(\texttt{radius})}\right| \leq 0.01$$

(b) and the cost is minimized; that is,

$$\mathbf{fx}\tau^* = \underset{\mathbf{fx}\tau \text{ satisfies (a)}}{\operatorname{argmin}} \texttt{cost}(\mathbf{fx}\tau(\texttt{mypi}, \texttt{radius}, \texttt{t}, \texttt{area}))$$

We use this example to illustrate the trade-off between cost and error and how a human might use trial and error to discover the correct wordlengths. We vary the wordlength of the variables. The integer wordlength is selected to avoid overflow and the remaining bits are used for fractional wordlength.

Case 1 (Figure 1): $WL = 12$ for all variables. $\mathbf{fx}\tau(\texttt{mypi}) = \langle 0, 2, 10\rangle, \mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 11\rangle, \mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 10\rangle, \mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 8\rangle$. Cost is 179.80.

Case 2 (Figure 2): $WL = 16$ for all variables. $\mathbf{fx}\tau(\texttt{mypi}) =$

$\langle 0, 2, 14\rangle$, $\mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 15\rangle$, $\mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 14\rangle$, $\mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 12\rangle$. Cost is 316.20.

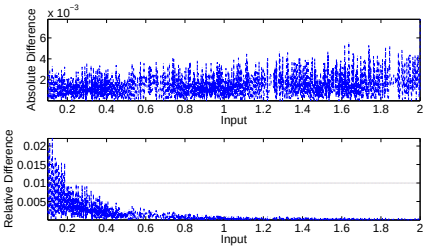


Fig. 1: $\texttt{WL} = 12$. Error threshold at 0.01 is violated.

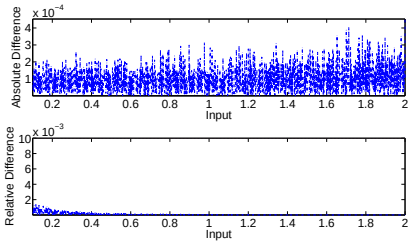


Fig. 2: $\texttt{WL} = 16$. Error threshold at 0.01 is not violated.

As we will show in the next section, our approach computes fixed-point types that meet the accuracy threshold and yield a cost of only 104.65, which, while being less than the cost in Case 1, satisfies the correctness criterion like Case 2. In the following section, we discuss our automated approach to solve this problem.

4 Our Approach

A central idea behind our approach, `swati` is to identify a small set of *interesting* inputs $S(X)$ using testing from the input domain $Dom(X)$ such that the optimal implementation found using induction that satisfies the correctness condition for the inputs in $S(X)$ will be optimal and correct for all inputs in the given input domain $Dom(X)$.

Procedure 3 Overall Synthesis Algorithm: `swati`

Input: Floating-point program F_{fp}, Fixed-point program F_{fx} with fixed-point variables T, Domain of inputs Dom, Error function Err, maximum error threshold `maxError`, Cost Model `cost`, maximum wordlengths $\texttt{WL}_{max}$

Output: Fixed-point type $\mathbf{fx}\tau$ for variables T or INFEASIBLE

S^0 = random sample from Dom, $Bad^0 = S^0$, $i = 0$
while $Bad^i \neq \emptyset$ **do**
  $i = i + 1$, $S^i = S^{i-1} \cup Bad^{i-1}$, $\mathbf{fx}\tau^i = \texttt{optInduce}(F_{fp}, F_{fx}, Dom, Err, \texttt{maxError}, \texttt{cost}, \texttt{WL}_{max}, S^i)$
  if $\mathbf{fx}\tau^i = \perp$ **then**
    return INFEASIBLE
  end if
  $Bad^i = \texttt{testErr}(F_{fp}, F_{fx}, \mathbf{fx}\tau^i, Dom, Err, \texttt{maxError})$
end while
return $\mathbf{fx}\tau^* = \mathbf{fx}\tau^i$

The top-level synthesis algorithm is presented in Procedure 3. $\texttt{WL}_{max}$ is an upper bound on wordlengths beyond which it is non-optimal to use the fixed-point version.

The algorithm starts with a randomly selected set of examples S^0 from the given input domain. Then, a fixed-point implementation that satisfies the accuracy condition for each of these inputs and is of minimal cost is synthesized using the routine `optInduce`. If no such implementation is found, the algorithm reports INFEASIBLE. Otherwise, the testing routine `testErr` checks whether the implementation fails the correctness condition for any input. If so, a set of inputs Bad^i on which the implementation violates the correctness condition are added to the set S^i used for synthesis, and the process is repeated. If the correctness condition is satisfied, the resulting fixed-point types are output. In the rest of this section, we describe the main components of our approach in detail, including the theoretical result.

4.1 Synthesizing Optimal Types for a Finite Input Set

The `optInduce` function (see Procedure 4) is used to obtain optimum fixed-point types such that the fixed-point program with these types satisfies the correctness condition for a finite input set S and has minimal cost. First, the floating-point program F_{fl} is executed for all the inputs in the sample S and the range of each variable t_i as well as its `Signedness` is recorded by the functions `getRange` and `isSigned` respectively. Then, the integer wordlength `IWL` sufficient to represent the computed range is assigned to each variable t_i and the `Signedness` is 1 if the variable takes both positive and negative values, and 0 otherwise. If the fixed-point program with maximum wordlengths $\texttt{WL}_{max}$ fails the correctness condition, we conclude that the synthesis is not feasible and return $\perp$. If not, we search for the wordlength with minimum cost satisfying the correctness condition using our optimization oracle $\mathcal{O}_S$. The result is used to compute the fractional wordlengths, and the resulting fixed-point types are returned.

More precisely, $\mathcal{O}_S$ solves the following optimization problem over $\mathbf{fx}\tau$: `Minimize cost`$(\mathbf{fx}\tau)$ `s.t.`

$$\bigwedge_{x \in S} Err(F_{fx}(x, \mathbf{fx}\tau), F_{fl}(x)) \leq \texttt{maxError} \tag{1}$$

Let us reflect on the nature of the above optimization problem. The overall synthesis algorithm might make several calls to $\mathcal{O}_S$ for solving the optimization problem for different sets of inputs and hence, $\mathcal{O}_S$ must be a fast procedure. But it is a discrete optimization problem with a non-convex constraint space, a problem class that is known to be computationally hard [7]. This rules out any computationally efficient algorithm to implement $\mathcal{O}_S$ without sacrificing correctness guarantees. Since the space of possible types grows exponentially with the number of variables, brute-force search techniques will not scale beyond a few variables. Satisfiability solvers can also not be directly exploited to search for optimal wordlengths since the existential quantification is over the types and not the variables. The arithmetic operators have different semantics when operating on operands with different types and hence, the only way to encode this search problem as a satisfiability problem is to case-split *exhaustively* on all possible types (word-lengths), where each case encodes the fixed-point program with one possible type. The number of such cases is exponential in the number of the variables in the program under synthesis and hence, SAT problems will be themselves exponentially large in size. Further, one would need to invoke SAT solvers multiple times in order to optimize the cost function. Thus, satisfiability solving would be a wrong choice to address this problem. Further, the space of possible types is also not totally ordered

and hence, binary search like techniques would also not work. For a binary search like technique to work, we will need to define a domination ordering over the types which has three properties. Firstly, it is a total ordering relation. Secondly, if a particular type assignment satisfies the correctness condition for all inputs then all dominating types satisfy the correctness condition for all inputs. Thirdly, the cost function is monotonic with respect to the domination ordering relation. In general, this may not be feasible for any given floating-point program and cost function. Hence, we implement $\mathcal{O}_S$ using a greedy procedure `getMinCostWL` presented in Procedure 5.

Procedure 4 Optimal Fixed-Point Types Synthesis: `optInduce`

Input: Floating-point program F_{fp}, Fixed-point program F_{fx} with fixed-point variables T, Domain of inputs Dom, Error function Err, maximum error threshold `maxError`, Cost Model `cost`, max wordlengths $\texttt{WL}_{max}$, Input S
Output: Optimal wordlengths `WL` for inputs S or $\perp$
  for all fixed-point variable t_i in F_{fx} **do**
    $\texttt{IWL}(t_i) = \lceil \log(\texttt{getRange}(t_i, F_{fl}, S) + 1) \rceil$, $\texttt{Signedness}(t_i) = \texttt{isSigned}(t_i, F_{fl}, S)$
  end for
  if $\texttt{WL}_{max} < \texttt{IWL}$ **then**
    return $\perp$
  end if
  $\mathbf{fx}\tau = \langle \texttt{Signedness}, \texttt{IWL}, \texttt{WL}_{max} - \texttt{IWL} \rangle$
  if $Err(F_{fp}(x), F_{fx}(x, \mathbf{fx}\tau)) > \texttt{maxError}$ **then**
    return $\perp$
  end if
  $\texttt{WL} = \texttt{getMinCostWL}(F_{fp}, F_{fx}, Dom, Err, \texttt{maxError}, fxcost, \texttt{WL}_{max}, S^i, \texttt{IWL}, \texttt{Signedness})$
  return $\mathbf{fx}\tau = \langle \texttt{Signedness}, \texttt{IWL}, \texttt{WL} - \texttt{IWL} \rangle$

4.2 Verifying a Candidate Fixed-Point Program

In order to verify that the fixed-point program $F_{fx}(X, \mathbf{fx}\tau)$ satisfies the correctness condition, we need to check if the following logical formula is satisfiable.

$$\exists X \in Dom(X) \;\; Err(F_{fx}(X, \mathbf{fx}\tau), F_{fp}(X)) > \texttt{maxError} \tag{2}$$

If the formula is unsatisfiable, there is no input on which the fixed-point program violates the correctness condition.

For arbitrary (possibly non-linear) floating-point and fixed-point arithmetic operations, it is extremely difficult to solve such a problem in practice with current constraint solvers. Instead, we use a novel optimization-based approach to verify the candidate fixed-point program. The intuition behind using an optimization-based approach is that the error function is continuous in the inputs or with very few discontinuities [17, 4], and hence, optimization routines can easily find inputs which maximize error function by starting from some random input and gradually adjusting the output to increase the value of the error function. The optimization oracle $\mathcal{O}_V$ is used to maximize the error function $Err(F_{fx}(X, \mathbf{fx}\tau), F_{fp}(X))$ over the domain $Dom(X)$. If there is no input $X \in Dom(X)$ for which the error function exceeds `maxError`, the fixed-point program is correct and we terminate. Otherwise, we obtain an example input on which

Procedure 5 `getMinCostWL`

Input: Floating-point program F_{fp}, Fixed-point program F_{fx} with fixed-point variables T, Domain of inputs Dom, Error function Err, maximum error threshold $maxErr$, Cost Model `cost`, max wordlengths $\texttt{WL}_{max}$, Input S
Output: Optimal wordlengths `WL`

```
valcandWL = {WL_max}
while valcandWL is not empty do
    WL = argmin_{vcWL ∈ valcandWL} cost(vcWL), fxτ = ⟨Signedness, IWL, WL − IWL⟩, candWL = ∅,
    valcandWL = ∅
    for all fixed-point variable t_i in F_fx do
        WL^{i−}(j) = WL(j) ∀j ≠ i, WL^{i−}(i) = WL(i) − 1, WL^{i+}(j) = WL(j) ∀j ≠ i, WL^{i+}(i) =
        WL(i) + 1, candWL = candWL ∪ {WL^{i−}, WL^{i+}}
    end for
    for all cand in candWL do
        candfxτ = ⟨Signedness, IWL, candWL − IWL⟩
        if Err(F_fp(x), F_fx(x, cand)) ≤ maxErr ∀x ∈ S
        and cost(candfxτ) < cost(fxτ) then
            valcandWL = valcandWL ∪ {cand}
        end if
    end for
end while
return fxτ
```

the fixed-point program violates the correctness condition. Multiple inputs can also be generated where they exist.

In practice, with the current state-of-the-art optimization routines, it is difficult to implement $\mathcal{O}_V$ to find a global optimum. Instead, we use a numerical optimization routine based on the Nelder-Mead method [19] which can handle arbitrary non-linear functions and generates local optima. Procedure 6 defines `testErr` which invokes the Nelder-Mead routine (indicated by "$\mathrm{argmaxlocal}$"). This routine requires one to supply a starting value of X, which we generate randomly. To find multiple inputs, we invoke the routine from different random initial points and record all example inputs on which the fixed-point program violates the correctness condition. Since a global optimum is not guaranteed, we repeat this search `maxAttempts` times before declaring that the fixed-point program is correct.

The following theorem summarizes the correctness and optimality guarantees of our approach. Proof is presented in extended version [12].

Theorem 1. *The synthesis procedure presented in Procedure 3 is guaranteed to synthesize the fixed-point program which is of minimal cost and satisfies the correctness condition for accuracy if optimization oracles $\mathcal{O}_S$ and $\mathcal{O}_V$ find globally-optimal solutions (when they exist).*

5 Experiments

Apart from the running example, we present case studies from DSP and control systems to illustrate the utility of the presented synthesis approach. Our technique was implemented in Matlab, and Nelder-Mead implementation available in Matlab as

Procedure 6 Verification Routine `testErr`

Input: Floating-point program F_{fp}, Fixed-point program F_{fx}, Fixed-point type $\mathbf{fx}\tau$, Domain of inputs Dom, Error function Err, maximum error threshold `maxError`

Output: Inputs Bad on which F_{fx} violates correctness condition

$Bad = \emptyset$
while $i \leq$ `maxAttempts` **do**
$i = i + 1$, $X_0 =$ random sample from Dom, $X_{cand} = \underset{X}{\text{argmaxlocal}}(Err(F_{fp}(X), F_{fx}(X, \mathbf{fx}\tau)), X_0)$
if $Err(F_{fp}(X_{cand}), F_{fx}(X_{cand}, \mathbf{fx}\tau)) >$ `maxError` and $X \in Dom$ **then**
$Bad = Bad \cup \{X\}$
end if
end while

`fminsearch` function was used for numerical optimization. We use the Constantinides et al [6] cost model.

5.1 Running Example

We illustrate the synthesis approach (more details in [12]) presented in Section 4 using the running example. Our algorithm used 34 examples and needed 4 iterations. To evaluate our approach, we exhaustively simulated the generated fixed-point program on the given domain ($0.1 \leq$ `radius` < 2) at intervals of 0.0001. The result is presented in Figure 5.1.

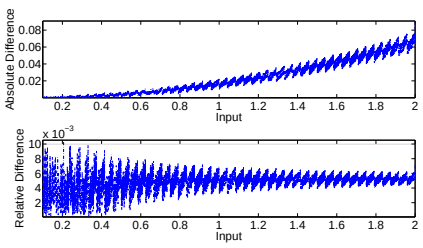


Fig. 3: Our Approach on Running Example

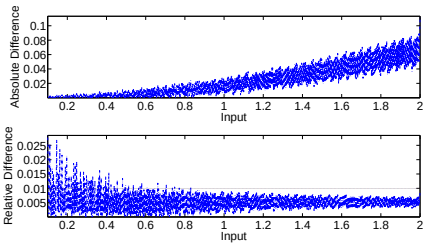


Fig. 4: Running Example Using Random Inputs.

As a point of comparison, we also show the result of synthesizing a fixed-point program using the `optInduce` routine with 100 inputs (3 times as many as our approach) selected uniformly at random (Figure 5.1). The horizontal line in the plots denotes the maximum error threshold of 0.01 on the relative difference error function. The cost of the fixed-point program synthesized with random sampling is 89.65, and the fixed-point types of the variables are $\mathbf{fx}\tau(\texttt{mypi}) = \langle 0, 2, 3\rangle$, $\mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 8\rangle$, $\mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 10\rangle$ and $\mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 8\rangle$. Notice, however, that it is incorrect for a large number of inputs. In contrast, the cost of the implementation produced using our technique is 104.65, and the fixed-point types of the variables are $\mathbf{fx}\tau(\texttt{mypi}) = \langle 0, 2, 3\rangle$, $\mathbf{fx}\tau(\texttt{radius}) = \langle 0, 1, 9\rangle$, $\mathbf{fx}\tau(\texttt{t}) = \langle 0, 2, 11\rangle$ and $\mathbf{fx}\tau(\texttt{area}) = \langle 0, 4, 10\rangle$.

5.2 Infinite Impulse Response (IIR) Filter

The first case study is a first-order direct form-II IIR filter (see extended version [12] for details). We use our synthesis technique to discover the appropriate fixed-point types of the coefficients of the filter. The input domain used in synthesis is $-2 < input < 2$. The correctness condition for accuracy is to ensure that the relative error between the floating-point and fixed-point program is less than 0.1.

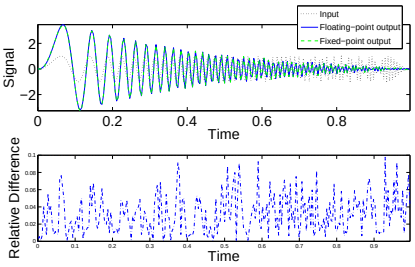


Fig. 5: IIR Filter

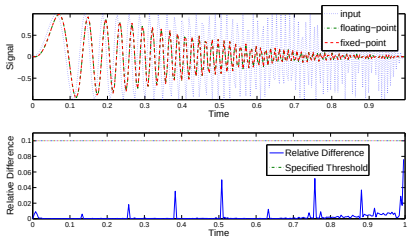


Fig. 6: FIR Filter

In order to test the correctness of our implementation, we feed a common input signal to both the IIR filter implementations: floating-point version and the fixed-point version obtained by our synthesis technique. The input signal is a linear chirp from 0 to $\frac{Fs}{2}$ Hz in 1 second.

$$input = (1 - 2^{-15}) \times sin(\pi \times \frac{Fs}{2} \times t^2)$$

where $Fs = 256$ and $t = 0$ to $1 - \frac{1}{Fs}$ and is sampled at intervals of $\frac{1}{Fs}$. Figure 5 shows the input, outputs of both implementations and the relative error between the two outputs. We observe that the implementation satisfies the correctness condition and the relative error remains below 0.1 throughout the simulation.

5.3 Finite Impulse Response (FIR) Filter

The second case study is a low pass FIR filter of order 4 with tap coefficients $0.0346, 0.2405, 0.4499, 0.2405$ and 0.0346. The input domain, correctness condition and input signal to test the floating-point implementation and synthesized fixed-point program are same as the previous case study. Figure 6 shows the input, outputs of both implementations and the relative error between the two outputs. We observe that the implementation satisfies the correctness condition and the relative error remains below 0.1 throughout the simulation.

5.4 Field Controlled DC Motor

The next case study is a field controlled DC Motor. It is a classic non-linear control example from Khalil [13]. A detailed discussion of this example is presented in the

extended version [12]. The goal in this work was to find an optimal fixedpoint implementation of the control law computed mathematically for DC motor. The computed control law can be mathematically shown to be correct by designers who are more comfortable in reasoning with real arithmetic but not with finite precision arithmetic. Its implementation using floating-point computation also closely mimics the arithmetic in reals but the control algorithms are often implemented using fixed-point computation on embedded platforms. We use our synthesis technique to automatically derive a low cost fixed-point implementation of the control law computing input u. The input domain is $0 \leq i_a, i_f, \omega \leq 1.5$ where i_a is armature current and i_f is field current. The correctness condition for accuracy is that the absolute difference between the control input u computed by fixed-point program and the floating-point program is less than 0.1.

Figure 7 shows the simulation of the system using the fixed-point implementation of the controller and the floating-point implementation. This end-to-end simulation shows that fixed-point program generated by our technique can be used to control the system as effectively as the floating-point program. This illustrates the practical utility of our technique. Figure 8 plots the difference between the control input computed by the fixed-point program and the floating-point program. It shows that the fixed-point types synthesized using our approach satisfy the correctness condition, and the difference between the control input computed by the fixed-point and floating-point program is within the specified maximum error threshold of 0.1. The number of inputs needed in our approach was 127. In contrast, the fixed-point types found using 635(5X our approach) randomly selected inputs violate the correctness condition for a large number of inputs.

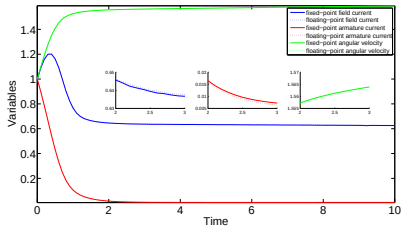


Fig. 7: DC Motor Using Floating-point and Fixed-point Controller. Fixedpoint and floating-point simulations almost overlap

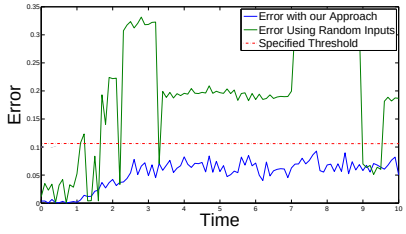
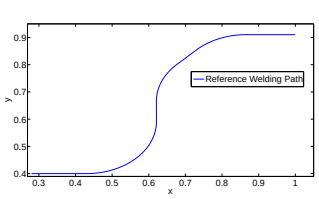


Fig. 8: DC Motor Error

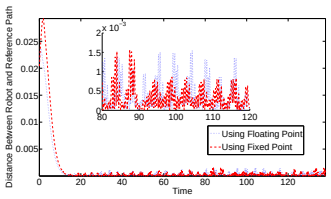
5.5 Two-Wheeled Welding Mobile Robot

The next case study is a nonlinear controller for a two-wheeled welding mobile robot (WMR) [3]. v and ω are the straight and angular velocities of the WMR at its center point which are the control parameters. Details of the robot model with equations of motion and the control law derivation is presented in extended version [12].

We use our synthesis technique to automatically synthesize fixed-point program computing both control inputs: v and ω. We require that the relative error for both controllers (v and ω) are less than 0.1. Figure 9(a) shows the reference line for welding



(a) Reference Welding Line



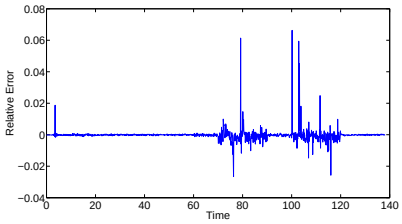
(b) Distance of WMR from Reference Line

Fig. 9: Welding Motor Robot

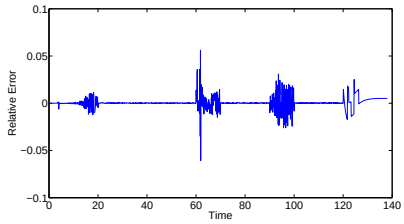
Table 1: Performance

Perf	IIR Filter	FIR Filter	DC Motor	WMR v	WMR ω
Runtime (s)	268	379	4436	2218	1720
# Iterations	5	4	8	7	4

and Figure 9(b) shows the distance of the WMR from the reference line as a function of time for both cases: firstly, when the controller is implemented as a floating-point program and secondly, when the controller is implemented as a fixed-point program synthesized using our technique. The robot starts a little away from the reference line but quickly starts tracking the line in both cases. Figure 10(a) and Figure 10(b) show the error between the floating-point controller and fixed-point controller for both control inputs: v and ω, respectively.



(a) Error in computing v



(b) Error in computing ω

Fig. 10: Welding Motor Robot

Performance: Table 1 summarizes the performance of our technique in the four case-studies.

6 Related Work

Previous techniques for optimizing fixed-point types are based on statistical sampling of the input space. These methods sample a large number of inputs and heuristically solve an optimization problem that minimizes implementation cost while ensuring that some correctness specification is met over the sampled inputs. The techniques differ in in the heuristic search method employed, in the measure of cost, or in how accuracy of fixed-point implementation is determined. Sung and Kum [22] use a heuristic search technique which starts with the minimum wordlength implementation as the initial guess. The wordlengths are increased one by one till the error falls below an acceptable threshold. Shi et al. [20] propose a floating-point to fixed-point conversion methodology for digital VLSI signal processing systems. Their approach is based on a perturbation theory which shows that the change to the first order is a linear combination of all the first- and second-order statistics of the quantization noise sources. Their technique works with general specification critera, as long as these can be represented as large ensemble averages of functions of the signal outputs. For example, they use mean-squared error (MSE) as the specification function. The cost of the implementation is a quadratic function. Monte Carlo simulation of a large number of input examples is used to formulate a quadratic optimization problem based on perturbation theory. In contrast, our specification requires that the accuracy condition holds for all inputs and not just on an average. Further, the cost function can be any arbitrary function for our technique and need not be quadratic. Perhaps most importantly, our technique does not rely on apriori random sampling of a large number of input values, instead using optimization to discover a small set of *interesting* examples which suffice to discover optimal fixed-point implementation. Purely analytical methods [21, 14] based on dataflow analysis have also been proposed for synthesizing fixed-point programs based on forward and backward propagation in the program's dataflow graph. The advantages of these techniques are that they do not rely on picking the right inputs for simulation, can handle arbitrary programs (with approximation), and can provide correctness guarantees. However, they tend to produce very conservative wordlength results. Inductive synthesis based on satisfiability solving has been previously used for synthesizing programs from functional specifications. These approaches [11, 9] rely on constraint solving in much the same way as we rely on optimization routines. However, these approaches only seek to find a correct program, without any notion of cost and optimization. An automated technique to minimize quantization error in control implementations is presented in [18]. They achieve this by modifying the LQR-LQG performance criterion and using the word-length as proxy for implementation cost. Our work predates [18] and was first reported in Chapter 4 of [10].

7 Conclusion

In this paper, we presented a novel approach to automated synthesis of fixed-point program from floating-point program by discovering the fixed-point types of the variables. The program is synthesized to satisfy the provided correctness condition for accuracy and to have optimal cost with respect to the provided cost model. We illustrated our approach on a set of case studies from digital signal processing and control systems.

Acknowledgement: This research was done when first author was at UC Berkeley. The research was supported by NSF grants CNS-0644436 and CNS- 0627734, the

FCRP/MARCO Multi-Scale Systems Center (MuSyC), Microsoft Research, Intel, and the Berkeley Fellowship for Graduate Studies from UC Berkeley.

References

1. ANSI/IEEE Std. 754-1985: IEEE standard for binary floating-point arithmetic.
2. Matlab: Fixed-point toolbox.
3. T.H. Bui, T.T. Nguyen, T.L. Chung, and S.B. Kim. A simple nonlinear control of a two-wheeled welding mobile robot. *Int. Journal of Control, Automation and Systems*, 1, 2003.
4. Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In *POPL '10*, pages 57–70, 2010.
5. Jonathan A. Clarke, Altaf Abdul Gaffar, George A. Constantinides, and Peter Y. K. Cheung. Fast word-level power models for synthesis of FPGA-based arithmetic. In *ISCAS*, 2006.
6. George A. Constantinides, Peter Y. K. Cheung, and Wayne Luk. Wordlength optimization for linear digital signal processing. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 22, No. 10*, pages 1432–1443, 2003.
7. R. Fletcher. *Practical Methods of Optimization*. J. Wiley, 1986.
8. David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991.
9. Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *PLDI*, pages 62–73, 2011.
10. Susmit Jha. *Towards Automated System Synthesis Using SCIDUCTION*. PhD thesis, EECS Department, University of California, Berkeley, Nov 2011.
11. Susmit Jha, Sumit Gulwani, Sanjit Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.
12. Susmit Jha and Sanjit A. Seshia. SWATI: Synthesizing wordlengths automatically using testing and induction. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-7.pdf`, 2013.
13. H.K. Khalil. *Nonlinear systems*. Macmillan Pub. Co., 1992.
14. Seehyun Kim, Ki-Il Kum, and Wonyong Sung. Fixed-point optimization utility for c and c++ based digital signal processing programs. *Circuits and Systems II: Analog and Digital Signal Processing, IEEE Transactions on*, 45(11):1455 –1464, nov 1998.
15. Mike Tien-Chien Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded DSP software. *VLSI*, 5(1):123 –135, march 1997.
16. E. Macii, M. Pedram, and F. Somenzi. High-level power modeling, estimation, and optimization. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 17(11):1061 –1079, nov 1998.
17. Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *RTSS '09*, pages 355–363, 2009.
18. Rupak Majumdar, Indranil Saha, and Majid Zamani. Synthesis of minimal-error control software. In *EMSOFT*, pages 123–132, 2012.
19. J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, January 1965.
20. Changchun Shi and Robert W. Brodersen. An automated floating-point to fixed-point conversion methodology. In *Proc. IEEE Int. Conf. on Acoust., Speech, and Signal Processing*, pages 529–532, 2003.
21. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bidwidth analysis with application to silicon compilation. *SIGPLAN Not.*, 35:108–120, May 2000.
22. Wonyong Sung and Ki-Il Kum. Simulation-based word-length optimization method for fixed-point digital signal processing systems. *Signal Processing, IEEE Transactions on*, 43(12):3087 –3090, dec 1995.
23. Randy Yates. Fixed-point arithmetic. In *Technical Reference Digital Signal Labs*, 2009.