On the Generation of Precise Fixed-Point Expressions

Eva Darulova
EPFL
eva.darulova@epfl.ch

Viktor Kuncak
EPFL
viktor.kuncak@epfl.ch

Rupak Majumdar
MPI-SWS
rupak@mpi-sws.org

Indranil Saha
UCLA
indranil@cs.ucla.edu

ABSTRACT

Several problems in the implementations of control systems, signal-processing systems, and scientific computing systems reduce to compiling a polynomial expression over the reals into an imperative program using fixed-point arithmetic. Fixed-point arithmetic only approximates real values, and its operators do not have the fundamental properties of real arithmetic, such as associativity. Consequently, a naive compilation process can yield a program that significantly deviates from the real polynomial, whereas a different order of evaluation can result in a program that is close to the real value on all inputs in its domain.

We present a compilation scheme for real-valued arithmetic expressions to fixed-point arithmetic programs. Given a real-valued polynomial expression t, we find an expression t' that is equivalent to t over the reals, but whose implementation as a series of fixed-point operations minimizes the error between the fixed-point value and the value of t over the space of all inputs. We show that the corresponding decision problem, checking whether there is an implementation t' of t whose error is less than a given constant, is NP-hard. We then propose a solution technique based on genetic programming. Our technique evaluates the fitness of each candidate program using a static analysis based on affine arithmetic. We show that our tool can significantly reduce the error in the fixed-point implementation on a set of linear control system benchmarks. For example, our tool found implementations whose errors are only one half of the errors in the original fixed-point expressions.

Categories and Subject Descriptors

D.2.10 [**Software**]: Software Engineering—*Design-Methodologies*

Keywords

Fixed-point arithmetic, genetic programming, synthesis, stochastic optimization, embedded control software

1. INTRODUCTION

Many algorithms in controls and signal processing are naturally expressed using real arithmetic. A direct implementation of these algorithms using floating-point computation requires either floating-point co-processors or software-based emulation of floating-point capabilities. Unfortunately, in many embedded domains where controls and signal processing applications are commonly used, the cost and power consumption of co-processors or the inefficiency of software emulation are unacceptable. Thus, these algorithms are usually implemented using a *fixed-point* equivalent of the original algorithm. A lot of research has gone into providing semi-automated compilation support from floating-point to fixed-point implementations [8, 22, 20, 5, 25]. The primary concern in these works has been bitwidth allocation: finding out the number of bits to allocate for the integral and the fractional parts of each real variable, so that the resulting implementation does not lose too much precision and the fixed-point variables do not overflow.

Even with an optimal bitwidth allocation, the precision of a fixed-point computation can depend on the order of evaluation of arithmetic operations. Since fixed-point arithmetic is not associative, and multiplication does not distribute over addition, the order in which a real polynomial is evaluated can cause differences in the error of the computation. These differences can indeed be significant: we show for one of our control system benchmarks that the error between two possible evaluation orders can be 2×: ranging from 0.00139 for the more precise expression to 0.00311 for the least precise one. Since the performance of controllers depends on the error introduced in the controller output, this difference can be significant. However, optimizing the error in the evaluation has received much less attention in fixed-point compilation, and has been limited to peephole optimizations (such as removing redundant shift operations locally) [1].

We present a technique to synthesize a fixed-point implementation for a given real-valued specification. Our synthesis method chooses the evaluation order of arithmetic operations to minimize the computation error. Given a real-valued arithmetic expression t, we aim to find a fixed-point implementation t', such that (1) the expressions t and t' are equivalent when interpreted over reals, and (2) the error between the real value and the fixed-point value computed by t' is minimal over all other fixed-point implementations equivalent to t. We show that the decision problem of finding an evaluation order that minimizes the error bound between the specification and the implementation is NP-hard, so a tractable *complete* search algorithm is unlikely.

Our technique is therefore based on a heuristic search implemented through genetic programming (GP) [26]. We use the mutation and crossover operations of genetic programming to generate new sub-expressions. To evaluate the fitness of a proposed solution, we use a static analysis based on affine arithmetic to compute an upper bound on the error. The objective of the search is to minimize the upper bound computed by the static analysis.

While our static analysis only computes an upper bound, we show, through extensive simulations, that the statically-computed upper bounds are proportional to the actual errors observed by simulations. We can thus use the less expensive upper bounds to compare two expressions with respect to precision.

We have implemented our technique and we have evaluated it on a set of control application benchmarks. Our experiments demonstrate that our technique is adequate in finding good fixed-point implementations for linear controllers. For non-linear computations we encounter limitations in using static analysis based on affine-arithmetic, but our search method works with any technique to estimate variable ranges, so further improvements in this area can be incorporated into our approach.

2. PRELIMINARIES

In this section we provide some background on the fixed-point representation of real numbers and genetic programming. As our benchmarks are mostly from the control engineering domain, we also provide a brief introduction to controller implementations.

In the rest of the paper, an *expression* denotes an arithmetic expression generated by the following grammar:

$$t ::= v \mid x \mid t_1 + t_2 \mid t_1 - t_2 \mid t_1 * e_2 \mid t_1/t_2$$

where v and x are rational constants or variables, respectively.

2.1 Fixed-point Representation

In a fixed-point implementation of an expression, all the variables and constants in the expression are assigned a fixed-point representation. A *fixed-point representation* of a real number is a triple $\langle s, v, w\rangle$ consisting of a *sign bit indicator* $s \in \{\mathtt{1}, \mathtt{0}\}$ (for *signed* and *unsigned*), a *length* $v \in \mathbb{N}$, and a *length of the fractional part* $w \in \mathbb{N}$. The length of the integer part is $v - w - 1$. Intuitively, a real number is represented using v bits, of which w bits are used to store the fractional part. A given integer w and a positive integer $v > 0$ determine a finite set $FX(v, w)$ of representable rational numbers. A variable with a fixed-point type is represented in a program as a v-bit integer. The fixed-point implementation of an expression then consists of assigning fixed-point representations to all input variables and intermediate results, i.e. to each node in the expression abstract syntax tree (AST). An arithmetic operation on two fixed-point variables involves integer arithmetic and shifting of bits. For more details on the semantics of fixed-point arithmetic, the readers are referred to [2].

Definition: **Worst-case error.** Assume a fixed-point implementation of an expression t. Given the values of variables, we define the expression error as $|t_r - t_f|$ where t_r is the value of t computed in real numbers, and t_f the value computed by the fixed-point implementation. Given the intervals for input variables of t, the *worst-case error* for a fixed-point implementation of t is the maximum over all expression errors where the values of variables range over the fixed point representable values from the given intervals.

2.1.1 Best fixed-point implementation

An operation using real arithmetic may have different fixed-point implementations depending on how many bits are allocated to hold the integer part and the fraction part of the variables and intermediate results. Allocating fewer bits than required to hold the integer part may lead to overflow. On the other hand, if more than the required number of bits are allocated to the integer part, the quantization error increases due to assigning fewer bits to the fractional part. When we compare the fixed-point implementations of different expressions, we consider their best possible implementation, which we define next. Let us fix the number of bits to be v for the representation of every variable in the implementation of an expression. Then we define the best fixed-point implementation as follows:

Definition: **Best fixed-point implementation for given intervals.** For a given v and given intervals for the variables and intermediate results, an implementation I is called the *best fixed-point implementation*, if for every input variable or intermediate result that takes values from an interval $[r^{min}, r^{max}]$, the fixed-point representation is given by $\langle 1, v, w\rangle$, where $w = v - 1 - z$ and z, the number of integral bits, is given by

$$z = \lceil \log_2(\max(\mathsf{abs}(r^{min}), \mathsf{abs}(r^{max})))\rceil \tag{1}$$

For example, if the interval for a variable is [-35.55, 48.72], the representation for the variable in the best 16-bit fixed-point representation has $z = 6$ bits for the integer part, so it is given by $\langle 1, 16, 9\rangle$. For a constant $C = 0.0864$ where $z = -3$, the representation is given by $\langle 1, 16, 18\rangle$.

Note that the best fixpoint implementation depends on intervals assigned to intermediate nodes. We say that intervals are tight if the intervals for internal nodes are as small as possible, which we can define by doing interval computation and fixed point allocation in parallel, as follows. Suppose we have assigned tight intervals S_1 and S_2 for sub-trees and that the operation is $*$. Let $S = \{x_1 * x_2 \mid x_1 \in S_1, x_2 \in S_2\}$ and let z and w be given by (1) taking $r^{min} = \inf(S)$ and $r^{max} = \sup(S)$. We then require the interval assigned to the node to be $[\mathrm{roundF}(r^{min}, v, w), \mathrm{roundF}(r^{max}, v, w)]$ where roundF denotes the rounding used in fixed-point computations when the representation is $(1, v, w)$.

A property of the above definitions is that, in the special case when the input intervals have lower bound equal to upper bound and are representable as fixed point numbers, then the tight intervals for intermediate nodes also have their lower bounds equal to their upper bounds, and are equal to the values of the sub-expression when evaluated in fixed-point arithmetic.

2.2 Control Systems

Control systems are developed to control certain behaviors of physical systems. Physical systems are typically modeled by differential equations:

$$\frac{d}{dt}x = f(x, u) \tag{2}$$

in which the curve $x : \mathbb{R} \to \mathbb{R}^n$ describes how the physical quantities of interest change over time. At each time instant

$t \in \mathbb{R}$, $x(t)$ is a vector in $\mathbb{R}^n$ containing the values of physical quantities such as positions, velocities, temperatures, pressures, etc. A controller of the form $u = k(x)$ is designed to control the evolution of the physical variables.

In this paper we mostly focus on linear control systems. To develop a linear control system, the behavior of the physical system is first approximated by linear differential equations, and then the differential equations are discretized based on a suitably chosen sampling time. A discrete-time linear time-invariant system is given by:

$$\begin{cases} x[r+1] = A_\tau x[r] + B_\tau u[r], \\ y[r] = Cx[r], \end{cases} \tag{3}$$

where $x \in \mathbb{R}^n$, $u \in \mathbb{R}^m$ and $y \in R^p$ represent the state, control input and output of the control system.

The controller requires the full state x to compute the control signal. However, the full state of the plant is not generally available to the controller, only output y of the system is available. Hence, the control input

$$u[r] = -K\widetilde{x}[r] \tag{4}$$

is computed based on an estimation $\widetilde{x}$ of the state x. The matrix K is called the feedback gain of the controller. The estimation $\widetilde{x}$ can be constructed using the observer dynamic [10]:

$$\widetilde{x}[r+1] = (A_\tau - B_\tau K - LC)\widetilde{x}[r] + Ly[r]. \tag{5}$$

The matrix L is called the gain of the observer.

Equation (5) together with Equation (4) provides the realization of the controller.

2.3 Genetic Programming

Genetic algorithms are heuristic search algorithms inspired by natural evolution. The algorithm evolves a population of candidate solutions by repeating the following steps for each new generation of solutions: from two candidates selected from the current generation new solutions are created by mutation and crossover whose quality is evaluated by a user-defined fitness function.

The candidate solutions are usually represented by strings so that these operations mimic closely natural evolution. Candidates for mutation and crossover can be selected for example by tournament selection where a fixed number of candidates is chosen at random and the one with the highest fitness is selected as the final candidate.

Note that the problem domain can be very complex and that it does not need to have a gradient for guiding the search. Genetic programming [26] is a variant of a genetic algorithm that performs the search over computer programs instead of strings. Mutation and crossover operators are thus defined on abstract syntax trees (ASTs).

3. EXAMPLE

We motivate the problem using a controller for a batch reactor processor [27]. The computation of a state of the controller is given by the following expression:

$$\begin{aligned} &(-0.0078) * \mathtt{st}_1 + 0.9052 * \mathtt{st}_2 \ + (-0.0181) * \mathtt{st}_3 + \\ &(-0.0392) * \mathtt{st}_4 \ + (-0.0003) * \mathtt{y}_1 + 0.0020 * \mathtt{y}_2 \end{aligned} \tag{6}$$

where $\mathtt{st_i}$ is an internal state of the controller and $\mathtt{y_i}$ is an input to the controller. There are additional similar expressions to compute the other states and the outputs of the controller.

```
tmp0 = ((-327161 * st1) >> 18)
tmp1 = ((296621 * st2) >> 15)
tmp2 = ((tmp0 + (tmp1 << 4)) >> 4)
tmp3 = ((-189791 * st3) >> 16)
tmp4 = (((tmp2 << 4) + tmp3) >> 4)
tmp5 = ((-205521 * st4) >> 15)
tmp6 = (((tmp4 << 4) + tmp5) >> 4)
tmp7 = ((-201331 * y1) >> 22)
tmp8 = (((tmp6 << 4) + tmp7) >> 4)
tmp9 = ((167771 * y2) >> 19)
tmp10 = (((tmp8 << 4) + tmp9) >> 4)
return tmp10
```

Figure 1: A possible fixed-point implementation for the example expression.

Consider a fixed-point implementation of this controller. If we assume an input range of $[-10, 10]$ for all input variables and a uniform bit length of 16, each input variable gets assigned the fixed-point format $\langle 1, 16, 11 \rangle$. This means that of the 16 bits we use 1 bit to represent the sign of the number, 4 bits for the integer part ($10 < 2^4 = 16$), and the remaining 11 bits for the fractional part. The constant -0.0078 gets the format $\langle 1, 16, 22 \rangle$ ($0.0078 < 2^{16-1-22} = 2^{-7} = 0.0078125$). If we multiply `st1` now by -0.0078, the result will have 33 bits, which we fit into 16 bits by performing a right shift. Following the order of arithmetic operations in (6) gives a fixed-point arithmetic program shown in Figure 1.

The fixed-point arithmetic implementation of the controller can have a large roundoff error. For example, because of the representation, the input values can already have an error as large as 0.00049. These errors then propagate throughout the computation. For a specific implementation, such as the one above, we can compute an upper bound on the error using an affine arithmetic-based static analysis. For our example, the maximum absolute error bound is 3.9e-3. We can bound the error from below using simulation, where we run the floating-point and the fixed-point programs side-by-side on a large number of random inputs and compare the results. Note that this technique only gives us an under-approximation. Using this approach, we get a lower bound on the error of 3.06e-3.

One way to reduce the error is to increase the bit length. If we add one bit to each variable, we get a simulated maximum error of 1.51e-3, which is an improvement by about 50%. However, increasing bitwidths require implementing circuits with larger areas, and may not be feasible.

A different possibility is to use a different order of evaluation for the expression. As fixed-point arithmetic operations are not associative, two different evaluation orders for the same implementation can have significantly different absolute errors. Consider the following reordering of Equation (6):

$$\begin{aligned} &((0.9052 * \mathtt{st}_2) + (((\mathtt{st}_3 * -0.0181) \ + (-0.0078 * \mathtt{st}_1)) + \\ &\quad (((-0.0392 * \mathtt{st}_4) \ + (-0.0003 * \mathtt{y}_1)) + (0.002 * \mathtt{y}_2)))) \end{aligned} \tag{7}$$

When implemented using 16-bit fixed-point arithmetic, we find, using our static analysis, that the maximum error bound is 1.39e-03, which is an even larger improvement of 55%, without requiring any extra hardware.

Our approach is to search the space of possible implementations of an arithmetic expression to find one that has the minimum fixed-point implementation error bound. We search the space using *genetic programming* (GP). GP finds this expression by evolving a population of expressions

expression	simulated error
original	3.06e-3
worst rewrite	3.11e-3
additional bit	1.52e-3
best rewrite	1.39e-3
best found by GP	1.39e-3

Figure 2: Summary of absolute errors for different implementations

through selection of the best expressions with respect to the error, mutation and crossover. Figure 2 summarizes the worst-case error bounds for the different formulations of the expression. By exhaustively enumerating all possible rewrites, we see that the maximum error bounds can vary between approximately 1.39e-3 and 3.11e-3. That is, even for a relatively short example, the worst error bound can be over a factor of 2 larger than the best possible one. GP can find the optimal expression without an exhaustive enumeration and can do this at analysis time. This does not cost any additional hardware, thus we get the additional precision "for free".

4. GENERATING FIXED-POINT EXPRESSIONS

In this section we describe our algorithm to solve the problem of synthesizing minimal-error fixed-point programs. That is, given a real valued expression t we aim to find an expression t' that is mathematically equivalent to t and whose implementation in fixed-point arithmetic minimizes, among all equivalent expressions, the worst-case absolute error over all inputs in given ranges I:

$$\min_{\text{equivalent } t'} \quad \max_{x \in I} \left| t_r(x) - t'_f(x) \right|$$

Algorithm 1 gives an overview of our search procedure, whose steps we explain in the following paragraphs.

Algorithm 4.1:

```
Input: expression, input ranges
initialize population of 30 expressions

repeat for 30 generations
  generate 30 new expressions:
     select 2 expressions with tournament selection
     do equivalence-preserving crossover
     do equivalence-preserving mutation
     evaluate fitness (worst-case error bound)

Output: best expression found during entire run
```

The input to our algorithm is a real valued expression and ranges for its variables. Our tool initializes the initial population with 30 copies of this expression.

4.1 Why Genetic Programming?

It is in general not evident from an expression whether it is in a good form with respect to precision and exhaustively enumerating all possible formulations of expressions becomes impossible very quickly. For only linear expressions the number of possible orders of adding n terms modulo commutativity, which does not affect precision, is $(2n-3)!!$[1].

(1) (a + b) + c = a + (b + c)
(2) a + b = b + a
(3) (-a) + (-b) = -(a + b)
(4) (a * b) * c = a * (b * c)
(5) a * b = b *a
(6) (-a) * b = - (a * b)
(7) a * (-b) = - (a * b)
(8) 1/a * 1/b = 1/(ab)
(9) - (1/a) = 1/(-a)
(10) (a * b) + (a * c) = a * (b + c)
(11) (a * c) + (b * c) = (a + b) * c
(12) (a * b) + (c * a) = a * (b + c)
(13) (b * a) + (a * c) = (b + c) * a

Figure 3: Rewrite rules.

For our example from Section 3 with 6 terms there are already 945 expressions. For our largest benchmark with 15 terms there are too many possibilities to enumerate.

We thus need a suitable search strategy to find a good formulation of an expression among all the possibilities. We show in Section 7 that the problem of finding an expression whose worst-case error bound is minimal is NP-hard and that it amounts to minimizing the ranges of intermediate variables. Since the inputs for the expressions can, in general, be positive and negative, optimizing one subcomputation may lead to a very large intermediate sum in a different part of the expression. An algorithm that tries to find the optimal solution in a systematic way (e.g. dynamic programming) is thus unlikely to succeed. Our problem also does not have a notion of a gradient and it cannot be easily formulated in terms of inputs and outputs or contraints.

Genetic programming does not rely on any of these features, and its formulation as a search over program AST fits our problem very nicely.

4.2 Instantiating Genetic Programming

We need to instantiate the general genetic programming algorithm, since we start with an already (mathematically) correct program and attempt to find a program that is *numerically* as correct as possible (correctness is defined with respect to an evaluation of the expression in mathematical reals). Thus, our mutation and crossover operators need to generate expressions that are mathematically equivalent to the initial expression and the fitness function needs to quantify the numerical precision.

Mutation.

The mutation operator selects a random node in the expression AST and applies one of the applicable rewrite rules from Figure 3. The rules capture the usual commutativity, distributivity and associativity of real arithmetic. Some of these rules do not have an effect on the numerical precision by themselves, but are necessary to generate other rewrites of an expression. To keep the operations simple, we rewrite subtractions ($a - b \rightarrow a + (-b)$) and divisions ($a/b \rightarrow a * (1/b)$) before the GP run.

Crossover.

While maintaining mathematical equivalence is easy for the mutation operation, in the case of crossover it is not evident how to perform it efficiently in general. Given two trees t_1 and t_2 as candidates for the crossover, the genetic algorithm picks a random node in t_1, which is the root of the subtree s_1. The problem is then the following: find a subtree s_2 in t_2 that is mathematically equivalent to s_1 in an efficient way. Instead of implementing a general decision procedure,

[1] The number of full binary trees with n leaves is C_{n-1}, where C_n are the Catalan numbers. We can label each of the trees in n! ways. Taking into account commutativity gives: $\frac{C_{n-1} \cdot n!}{2^{n-1}}$.

we chose to do the following. At initialization, each subtree is annotated with a label that is the string representation of the expression at that subtree. During mutation, labels are preserved in the new generation as much as possible. For example, suppose we have the node $(a+b)+c$, with label `(a + b) + c`. We can apply mutation rule 1 to obtain $a+(b+c)$ but the label will remain `(a + b) + c`. Note that some of the mutation rules break equivalences (e.g. mutation rule 10), hence not all labels can be preserved. In that case we add a new label. During crossover, we then only need to check for identical labels. If labels match, it means that the subtrees come from the same initial subtree and hence are mathematically equivalent and we can exchange them in a crossover operation.

Parameters.

Our genetic programming pipeline has several parameters that can influence the results: the number of best individuals passed on to the next generation unchanged (elitism) (0, 2 or 6), the number of individuals considered during tournament selection (2, 4 or 6), and the probability of crossover (0.0, 0.5, 0.75 or 1.0). The most successful setting we found is with a tournament selection among 4 and an elitism of 2 while performing crossover every time, i.e. with probability of 1.0. Note, however, that even in the case of other settings, the improvements are still significant (on the order of 50%).

4.3 Fitness Evaluation

We use a static analysis based approach to compute the fitness of an expression. Our static analysis tool computes sound over approximations of the ranges of all variables and of the maximum absolute error of the corresponding fixed-point implementation.

Our tool uses affine arithmetic to compute the ranges of all intermediate values. From this we can determine the best possible fixed-point format and the quantization error at each computation step. Our approach is similar to [8], but we treat constants like normal variables and we do not discard higher order terms, which in the approach taken in [6] for floating-point arithmetic. The latter difference means that the error bounds we compute are sound with respect to real arithmetic.

If we are interested in proving that the roundoff errors stay within certain bounds, the computed bounds on the absolute errors need to be as tight as possible. Note that the main requirement on the analysis in our problem is slightly different. While tight bounds on errors are an advantage, what we need to know is the *relative precision* of our analysis tool. That is, we need to know whether the analysis tool is able to distinguish a better implementation from a less precise one. To see why this is different from the usual case, note that the analysis tool assumes worst-case errors at each computation step. It general, however, the worst-case errors will not be attained at all computation steps.

Thus, before using our analysis tool in a GP framework, we evaluate this property experimentally. We generate a number of random rewrites for an expression, for which we then obtain the actual errors by simulation. We present here the results for one linear and one nonlinear benchmark (batch controller, state 2 and rigid body, out 1 respectively). For 100 random different expression formulations, the ratio between the analyzed upper bound on the error and the simulated lower bound on the error has a mean of 1.29387

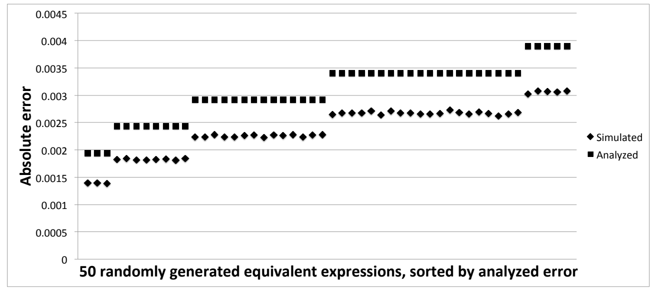

Figure 4: Comparison of analyzed upper bound and simulated lower bound on maximum errors for the linear benchmark batch processor (state 2).

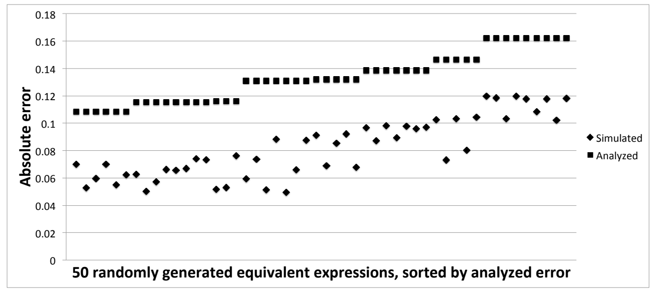

Figure 5: Comparison of analyzed upper bound and simulated lower bound on maximum errors for the nonlinear benchmark rigid body (out1).

and a variance of 0.00082 for the batch controller, state 2 benchmark and a mean of 1.66697 and a variance of 0.08315 for the rigid body, out 1 benchmark. Figures 4 and 5 show a direct comparison between the analyzed and simulated errors. In the linear case, the computed bounds on the errors are proportional to the actual errors, thus indicating a good relative precision. In the nonlinear case the correspondence is not as precise, however we expect it to be still sufficient for our purpose. The "more nonlinear" a computation becomes, the less precise we expect affine arithmetic to be.

5. OPTIMAL CONTROLLER SYNTHESIS

The controller for a discrete-time linear control system is given by Equation (5) and Equation (4). If we implement the controller using fixed-point arithmetic, we introduce additive error to the output of the controller. Thus the fixed-point implementation of the controller is given by:

$$\begin{cases} \widehat{x}[r+1] = (A_\tau - B_\tau K - LC)\widehat{x}[r] + Ly[r] + e_{q1}, \\ \widehat{u}[r+1] = -K\widehat{x}[r+1] + e_{q2}, \end{cases} \tag{8}$$

where $e_{q1} \in \mathbb{R}^n$ and $e_{q2} \in \mathbb{R}^m$. The vector $e = \begin{bmatrix} e_{q1} \\ e_{q2} \end{bmatrix}$ captures the implementation error of the controller.

One of the fundamental properties of a control system is *asymptotic stability*. It is possible to design a controller mathematically (finding the matrices K and L) such that the system in (3) is asymptotically stable with respect to origin [10], which intuitively means that the state of the plant will asymptotically converges to the origin. However, as shown in [2], in the presence of implementation error the state of plant can only be shown to converge asymptotically in a set around the origin. The set is called the *region of* *practical stability*. The following proposition formalizes the result. We use $\|x\|$ to represent euclidian norm of x.

PROPOSITION 1. *[2] Assume that the mathematical controller in* (4) *with the observer in* (5) *can render the plant in* (3) *asymptotically stable. If there exists a constant b such that* $\|e\| \leq b(e)$*, then the implementation* (8) *of the controller is guaranteed to render the state of the plant asymptotically to the set*

$$\mathcal{A}_y = \{y \in \mathbb{R}^p \mid \|y\| \leq \gamma_y b(e)\}, \tag{9}$$

where γ_y *is given by:*

$$\gamma_y = \max_{\theta \in [0,\ 2\pi[} \left\| [C \quad 0_{p\times n}] \left(\mathsf{e}^{i\theta} I_{2n} - G\right)^{-1} H \right\|. \tag{10}$$

where γ_y *is called the* $\mathcal{L}_2$*-gain of the control system. The matrices* G *and* H *are given by*

$$G = \begin{bmatrix} A_\tau & -B_\tau K \\ LC & A_\tau - B_\tau K - LC \end{bmatrix}, \quad H_2 = \begin{bmatrix} 0_{n\times n} & B_\tau \\ I_n & 0_{n\times m} \end{bmatrix}.$$

Controller synthesis has been traditionally performed by minimizing LQR and LQG costs [10]. Recently, Majumdar et al. [19] showed how to synthesize a controller co-optimizing both the LQR/LQG cost and the region of practical stability. They optimize a weighted linear combination of the two cost functions using a stochastic local search technique. The optimization problem is non-convex, and is solved using *particle swarm optimization* (PSO), a population-based stochastic optimization approach [15]. PSO iteratively solves an optimization problem by maintaining a population (or *swarm*) of candidate controllers, called *particles*, and moving them around in the search-space of possible controllers, trying to minimize the objective function.

In each step of PSO, given a new controller, a bound on the implementation error is estimated for a naive implementation. The value of the objective function is computed by taking the weighted sum of the LQR-LQG cost and this bound. Note that the implementation does not consider all possible expressions for a controller. If the controller is given by $K = [k_1,\ k_2, \ldots, k_n]$ and the state of the plant is denoted by $x = [x_1,\ x_2, \ldots, x_n]$, then the expression that is considered for the implementation of the fixed-point program is of the form $f = (((k_1x_1 + k_2x_2) + k_3x_3) + \ldots + k_nx_n)$. We refer to this expression as the *naive expression*. Note that the implementation of another expression may give a better bound on the error. The expression giving the least bound for the error is referred to as the *best expression*.

We call the implemented controller corresponding to the naive expression the *baseline implementation*. The naive expression can be passed through the genetic algorithm based rewriting method to get the best expression, and the corresponding controller implementation is called the *improved implementation*. Let us suppose that the controller gains synthesized by the PSO method in [19] is denoted by K. Let us assume that the bound on the error in its baseline implementation is b^b_K, and that for its improved implementation is b^i_K. Now there may exist another controller K' with the bounds on the error in its baseline implementation and improved implementation to be $b^b_{K'}$ and $b^i_{K'}$ respectively such that $b^b_K < b^b_{K'}$, and $b^i_{K'} < b^i_K$. This implies that if we use the method in [19] to implement the baseline controller and then employ the rewriting scheme to get the improved controller, we may not obtain the best possible controller implementation.

To achieve the best possible implementation for a controller, we take the following strategy. In every step of PSO, for a given controller, we start with the naive expression and apply our genetic-programming-based rewriting technique to find the best expression. We use this bound on the implementation error of the best expression in the objective function. The controller implemented using this new objective function is referred as the *optimal implementation*. In the next section, we show that such combination of search strategies substantially improves the guaranteed properties of the synthesized controllers, showing that techniques for synthesis of fixed-point programs have concrete benefits for practical controller synthesis.

6. EXPERIMENTS

We evaluate our technique on a number of benchmarks. Our benchmarks include a bicycle model [4], a DC motor position control [28], a pitch angle control [28], an inverted pendulum [28] and a batch reactor process [9]. The controllers for these system were taken from [19], which attempted to minimize the size of the region of practical stability by choosing a controller whose fixed point implementation has the best possible bound on the error among all controllers that stabilize the plant. To show the scalability of our tool we choose the example of a locomotive pulling a train car where the connection between the locomotive and the car is modeled by a spring in parallel with a damper [24]. By increasing the number of cars, we can increase the dimension of the system. We also consider a nonlinear controller for a rigid body [3] and the nonlinear B-splines functions [18]. Though most of our benchmarks are from the controller domain, nothing in our approach is actually domain specific.

Each benchmark consists of one expression and ranges for its input parameters. We wish to minimize the error on the one output value it computes over all possible inputs. Some of the benchmarks compute internal states of a controller (denoted with e.g. "state 1"). Since each state is computed with a different expression, we treat them here as separate benchmarks.

For all benchmarks we consider a fixed bit length, signed fixed-point format and truncation as the rounding mode. Table 6 lists the maximum absolute errors (computed by our analysis tool) for the best expressions found by GP for all our benchmarks.[2] The benchmarks are ordered approximately by complexity with the smaller linear benchmarks first and the nonlinear benchmarks at the end of the table. From the third column we can see that we get substantial improvements in precision of up to 70%.

6.1 Exhaustive Rewriting

For our smaller benchmarks (linear with up to 6 terms) we also generate all possible rewrites up to commutativity and determine tight bounds on the maximum errors by simulation. For this, we first automatically generate a fixed-point implementation, which we then evaluate on a number of random inputs. We use the floating-point code as reference, thus obtaining a lower bound on the error. Using a large enough number of random inputs (10 000 000), we

[2] The source code for our tool is avaiblable on reviewer request.

Benchmark	Original	Best (% of original)	Worst (% of original)	Added bit (% of original)
batch processor (out 1)	2.89e-3	0.91	1.37	0.51
batch processor (out 2)	7.20e-3	0.81	1.13	0.49
batch processor (state 1)	2.66e-3	0.52	1.02	0.50
batch processor (state 2)	3.06e-3	0.45	1.03	0.50
batch processor (state 3)	2.66e-3	0.50	1.16	0.49
batch processor (state 4)	2.24e-3	0.61	1.40	0.51
traincar 1 (out)	5.29e-5	0.78	1.02	0.60
traincar 1 (state 1)	1.02e-6	0.97	1.01	0.50
traincar 1 (state 2)	2.66e-7	0.71	1.02	0.52
traincar 1 (state 3)	2.04e-7	0.74	1.14	0.48

Figure 7: Best and worst absolute errors, determined by simulation.

Benchmark	err	$\frac{\text{orig.- no-cross}}{\text{orig.}}$	$\frac{\text{orig.- best}}{\text{orig.}}$	g
bicycle (out1)	2.66e-3	0.00	0.00	-
bicycle (state1)	2.53e-4	0.19	0.19	1
bicycle (state2)	1.82e-4	0.00	0.00	-
dc motor (out1)	1.06e-4	0.00	0.00	-
dc motor (state1)	2.77e-4	0.00	0.00	-
dc motor (state2)	3.75e-4	0.25	0.25	4
dc motor (state3)	1.27e-4	0.00	0.00	-
pendulum (out1)	8.09e-8	0.03	0.03	5
pendulum (state1)	5.13e-9	0.17	0.17	1
pendulum (state2)	6.11e-9	0.38	0.38	16
pendulum (state3)	5.14e-9	0.00	0.00	-
pendulum (state4)	4.97e-9	0.27	0.27	7
pitch angle (out1)	1.33e-7	0.18	0.18	4
pitch angle (state1)	4.26e-9	0.30	0.30	2
pitch angle (state2)	2.79e-9	0.00	0.00	-
pitch angle (state3)	3.81e-9	0.20	0.20	2
batch reactor (out1)	5.15e-4	0.00	0.00	-
batch reactor (out2)	1.28e-3	0.12	0.12	2
batch reactor (state1)	3.46e-4	0.15	0.15	1
batch reactor (state2)	2.77e-4	0.00	0.00	-
batch reactor (state3)	3.55e-4	0.26	0.26	2
batch reactor (state4)	4.11e-4	0.23	0.23	7
traincar 1 (out)	1.11e-4	0.09	0.09	2
traincar 1 (state 1)	1.98e-6	0.03	0.03	6
traincar 1 (state 2)	3.57e-7	0.25	0.25	16
traincar 1 (state 3)	2.79e-7	0.24	0.24	7
traincar 2 (out)	7.40e-5	0.09	0.09	19
traincar 2 (state 3)	1.23e-7	0.49	0.59	21
traincar 3 (out)	1.26e-3	0.13	0.13	7
traincar 3 (state 6)	1.32e-7	0.48	0.58	21
traincar 3 (state 7)	1.31e-7	0.43	0.53	17
traincar 4 (out)	9.34e-3	0.26	0.29	27
traincar 4 (state 1)	7.29e-8	0.73	0.73	19
traincar 4 (state 2)	7.34e-8	0.67	0.73	25
traincar 4 (state 3)	1.01e-7	0.66	0.60	14
traincar 4 (state 4)	6.96e-8	0.64	0.70	26
traincar 4 (state 5)	1.42e-7	0.61	0.68	26
traincar 4 (state 6)	1.67e-7	0.59	0.59	16
traincar 4 (state 7)	1.67e-7	0.56	0.56	13
traincar 4 (state 8)	1.38e-7	0.60	0.60	19
traincar 4 (state 9)	1.67e-7	0.47	0.47	7
bspline 1	2.29e-4	0.36	0.36	6
bspline 2	1.66e-4	0.52	0.52	4
rigid-body (out1)	1.08e-1	0.33	0.33	5
rigid-body (out2)	9.92e-1	0.20	0.20	15

Figure 6: Maximum absolute errors for the best expression found by GP with the settings elitism: 2, tournament selection: 4, with and without crossover (seed used: 4357). err is the analyzed error. g denotes the generation in which the solution is found.

obtain reasonably tight error bounds. Figure 7 shows the simulation results for the original formulation of the expression and the best and worst among all the possible rewrites. From the 10 benchmarks, 6 have an error in the original formulation that is about as bad as it gets. But we can also see that the best possible rewrite can improve the precision substantially. On the other hand, the expression may also be such that no matter how we rewrite it, the precision does not vary much, as is the case with the traincar 1, state 1 benchmark. This case, however, seems to be rather rare.

The other possibility of improving the precision is to increase the bit length. We do so in a minimal way by allowing one more bit for each intermediate variable, i.e. we increase the bit length by one and evaluate the precision with simulation. Note that in many cases, such a gradual increase may not be possible and one would have to add a whole word, e.g. go from 16 to 32 bits, with the associated hardware cost. Compile-time transformations, on the other hand, come "for free" and, as Figure 7 shows, can have an effect on the same order as the addition of a bit. In the case of longer benchmarks, the effect of rewriting can be even larger (see Figure 6), while the added bit always gains only about 50% of precision as compared to the original error.

6.2 Genetic Programming

Table 6 shows the results obtained for the most successful setting we have found. It also shows the results with crossover turned off. The comparison of these two columns suggests that crossover is helpful. We therefore expect that randomized local search techniques are not as effective as genetic programming, but they still produce useful reductions in the errors.

Optimality.

For the smaller benchmarks, GP always finds the same expressions with respect to the error bounds. Exhaustive enumeration has confirmed that the found expressions are indeed the optimal ones. For larger benchmarks, we do get improvements and we know of no technique to obtain better results.

Performance.

The runtimes of our GP algorithm depend in general mostly on the number of generations considered and the population size. Crossover only has a small effect on the overall runtime, but we have found that it provides the best results in the setting given above. In Table 8 we report the running times of our benchmarks with the default setting of 30 generations with a population size of 30 and the best GP settings we found.

Examples	Runtime (s)
batch (out1)	1.964
batch (state2)	2.735
traincar 1 (state 3)	6.358
traincar 2 (state 5)	9.794
traincar 3 (state 7)	15.388
traincar 4 (state 9)	17.228
rigid-body (out1)	1.394
rigid-body (out2)	2.698
bspline 1	2.234

Figure 8: Average runtimes of GP on selected benchmarks in seconds. Experiments were performed on a 3.5GHz Linux desktop with 16GB RAM.

Example	GP	Random	(Rnd-GP)/Rnd
traincar 4 (out)	0.00934	0.0103	0.09
traincar 4 (1)	7.29e-8	2.07e-7	0.65
traincar 4 (2)	7.34e-8	2.08e-7	0.65
traincar 4 (3)	1.01e-7	1.90e-7	0.47
traincar 4 (4)	6.96e-8	1.74e-7	0.60
traincar 4 (5)	1.42e-7	3.21e-7	0.56
traincar 4 (6)	1.67e-7	2.86e-7	0.42
traincar 4 (7)	1.67e-7	2.57e-7	0.35
traincar 4 (8)	1.38e-7	2.28e-7	0.39
traincar 4 (9)	1.67e-7	1.97e-7	0.15

Figure 9: Comparison of maximal errors between best expressions from GP and random search.

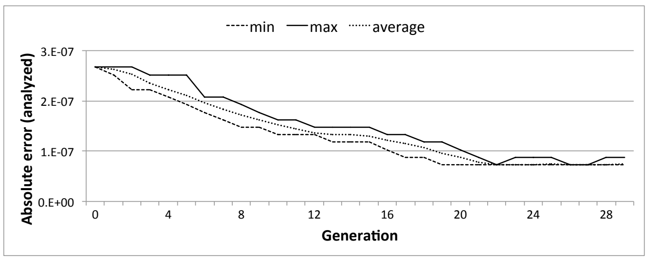

Figure 10: Evolution of errors across generations for the traincar 4 - state 1 example.

Control systems	*lub* of LQR cost		LQG cost	
	Baseline	Optimal	Baseline	Optimal
bicycle	$4.33{+}3\|x\|^2$	$4.33e{+}3\|x\|^2$	2.46e-2	2.47e-2
dc motor	$1.38e{+}3\|x\|^2$	$1.50e{+}3\|x\|^2$	3.67e+1	3.67e+1
pitch angle	$2.99e{+}6\|x\|^2$	$2.99e{+}6\|x\|^2$	1.80e-3	1.58e-3
pendulum	$5.35e{+}4\|x\|^2$	$5.35e{+}4\|x\|^2$	3.90e-1	3.90e-1
batch reactor	$2.23e{+}2\|x\|^2$	$2.23e{+}2\|x\|^2$	9.49e-2	9.08e-2

Figure 13: Least upper bound (*lub*) on the LQR cost, for a given initial condition x and the LQG cost for the baseline and the optimal implementations.

Efficiency improvement over random search.

For the expressions of the traincar 4 controller (15 terms) we also compare the results from the GP algorithm against a random search. This experiment is performed by generating 900 random and unique rewrites of the original expression, and comparing the best seen expressions against each other. Since we run the GP algorithm for 30 generations with a population of 30, we can see at most 900 unique expressions. As the results in Table 9 confirm, the GP based search is more effective than a random one. The third column shows the relative difference between the errors. Thus, many times the GP found expression is by over 50% more accurate than a randomly found one. That GP does not perform a random search can also be seen on the evolution of the population in Figure 10 for one benchmark (traincar 4, state 1). The plot shows the best, worst and the average errors of the expressions in each generation. The convergence to a low-error expression is clear.

6.3 Performance Enhancement in Control Systems

We implemented the algorithm presented in Section 5 in Matlab. The algorithm incorporates genetic algorithm based expression rewriting in the search for the optimal controller using PSO. We use a PSO function in Matlab from [7]. We have used the same setup for PSO as used in [19]. In Table 11 we provide the synthesized controllers and the time required to synthesize them. The synthesis experiments were done on a laptop running Mac OS X version 10.7.4 with 2 GHz Intel Core i7 CPU and 8GB 1600MHz DDR3 Memory.

In Table 12, we present the size of the region of practical stability for the baseline, improved and the optimal controllers for different benchmark systems and also the percentage improvement in the size of the region for the improved and the optimal controllers. The baseline and the improved implementations are based on the controllers provided in [19], and the optimal implementations are based on the controllers synthesized using the algorithm presented in Section 5. Note that the region of practical stability for the baseline implementation varies from the result provided in [19]. For example, for bicycle, the size of the region was presented as 0.0513 ignoring the effect of disturbance and measurement noise, the corresponding figure is 0.0785 in our experiment. This discrepancy is due to the fact that we use a different method to estimate the bound on the error in the fixed-point implementation. Our abstract interpretation based error estimation method is an order of magnitude faster than the mixed-integer linear programming approach in [19], which is apparent from the "time cost" column in Table 12. Even after incorporating the genetic programming based expression evaluation method in the synthesis process, our tool takes less time to synthesize a controller for all the benchmarks. Moreover, though our error estimation is less precise in comparison to that of [19] for 16 bit implementations, for 32 bit implementations (pitch angle and inverted pendulum) our estimation is significantly more precise.

The results in Table 12 show that we can improve the size of the region by rewriting the expression used in the baseline implementation. However, the improvement becomes significant when we incorporate the rewriting technique in the search method. Our results show that it is even possible to achieve more that 50% improvement in the synthesized controller with respect to the baseline controller. Table 13 shows the LQR/LQG cost of the baseline and the optimal controllers. The results show that in most of the examples, LQR and LQG costs do not degrade in the optimal controller with respect to the baseline controller. Only for the DC motor position example, the degradation in LQR cost is 8%. In a few instances, the LQG cost even improves.

Control systems	# bits	Synthesized gains		Time cost
		K	L	
bicycle	16	$[\ 3.0265e+0\ \ 1.2608e+1\ ]$	$[6.9088e-3\ \ 1.1135e-1\ ]^T$	51m36s
dc motor	16	[1.1760e-1 1.7400e-2 1.3300e-2]	$[$ 4.0400e-2 3.6720e-1 -1.2400e-2$]^T$	43m15s
pitch angle	32	[-1.2022e-1 4.2566e+1 1.0004e+0]	$[$ 3.2131e-4 2.1565e-5 1.8907e-3 $]^T$	1hr21m58s
pendulum	32	[-1.5362e+0 -2.0254e+0 1.6519e+1 2.7358e+0]	$\begin{bmatrix} 1.7000e\text{-}3 & 2.1000e\text{-}3 & 1.2000e\text{-}3 & 0.0000e\text{+}0 \\ 1.0000e\text{-}4 & 1.8000e\text{-}3 & 1.2200e\text{-}2 & 7.7000e\text{-}2 \end{bmatrix}^T$	38m43s
batch reactor	16	$\begin{bmatrix} 5.9434e\text{-}2 & 9.0617e\text{-}1 & 3.2788e\text{-}1 & 8.7115e\text{-}1 \\ -2.4646e\text{+}0 & -4.4966e\text{-}2 & -1.7086e\text{+}0 & 1.1691e\text{+}0 \end{bmatrix}$	$\begin{bmatrix} 7.6055e\text{-}2 & -1.8342e\text{-}3 & 2.9025e\text{-}2 & 3.0801e\text{-}2 \\ -1.1106e\text{-}2 & 2.2255e\text{-}2 & 3.9666e\text{-}2 & -9.2832e\text{-}4 \end{bmatrix}^T$	2hr45m35s

Figure 11: Synthesized gains and required time for synthesizing them.

Control systems	Region of Practical Stability			Improvement (%)	
	Baseline	Improved	Optimal	Improved	Optimal
bicycle	7.85e-02	7.70e-02	6.99e-02	1.93	10.96
dc motor	1.64e-02	1.44e-02	9.80e-03	12.14	40.24
pitch angle	1.08e-02	8.87e-03	5.15e-03	18.00	52.32
pendulum	3.11e-04	2.64e-04	2.51e-04	14.76	19.26
batch reactor	2.59e-01	2.24e-01	2.07e-01	13.31	20.08

Figure 12: Improvement in the size of region of practical stability for the improved and synthesized controllers

7. OPTIMAL FIXED-POINT PROGRAM SYNTHESIS PROBLEM IS NP-HARD

This section shows that given an arithmetic expression, the problem of finding a mathematically equivalent expression for which the computed worst-case error bound of the output of the fixed-point implementation is least is NP-hard. This justifies our use of a heuristic search method such as genetic programming. Though our algorithm supports operators '+', '-' , '*' or '/', we show the NP-hardness proof already for a problem that deals with the operator '+' alone.

For such an expression T, we define $E(T)$ as the *worst-case error bound of the best fixed-point implementation*, as follows.

Let the set of internal nodes of T be denoted by n_i and consider the best fixed-point implementation of T with tight intervals, as introduced in Subsection 2.1. As the worst case error e_i at node n_i we use

$$e_i = R(\max(\mathsf{abs}(r_i^{min}), \mathsf{abs}(r_i^{max}))))/2^{v-1}$$

where R is a function used to make the bound uniform. A possible sound choice for R include s $R(x) = 2^{\lceil \log_2 x \rceil}$, which, according to the Definition (1) makes the error equal to the value of the least significant bit 2^{-w}. Another, slightly more conservative choice, is $R(x) = 2^{1+\log_2 x} = 2x$, which we adopt here.

Minimum-Error Fixed-point Set Range Sum (MEFxRS): Let $X = \{x_1, \ldots, x_p\}$ denote a set of variables, $x_i \in \mathbb{R}$. Given an expression of the form $\sum_{i=1}^{p} x_i$, where each variable x_i can take value from a range $[r_i^{min}, r_i^{max}]$, find the ordering of the addition operations that yields the minimal worst-case error bound of the best fixed-point implementation of the expression.

Our objective is to show that the problem MEFxRS is NP-hard. Towards that end, we first define a simplified problem where we compute the sum of a set of integers (instead of intervals). Note that the numbers may be both positive and negative.

Minimum-Error Fixed-point Set Value Sum (MEFxVS): Let $X = \{v_1, \ldots, v_p\}$ denote a set of integers. Given an expression of the form $\sum_{i=1}^{p} v_i$, find out the ordering of the addition operations, that yields the minimal worst-case error bound at the output of the best fixed-point implementation of the expression.

It is straight-forward to show that an instance of a MEFxVS problem with values v_i can be reduced to a MEFxRS problem, for example, by letting $r_i^{min} = r_i^{max} = v_i$. In what follows, we show that MEFxVS is NP-hard.

To derive an expression for the worst-case error bound, we consider the AST of the expression, and in particular one internal node n_i representing a partial sum. Let the fixed-point value at n_i be c_i. To use our definition for the worst-case error bound, note that $r_i^{min} = r_i^{max} = c_i$. Then,

$$e_i = \frac{1}{2^{v-1}} R(|c_i|) = \frac{1}{2^{v-2}} |c_i|$$

Thus, at any internal node n_i, the error e_i is $\alpha|c_i|$ for $\alpha = 2^{-(v-2)}$. The errors at the leaf nodes are constant and we denote their sum by e_0. The worst-case error bound for the implementation tree T is thus given by $E(T) = e_0 + \alpha \sum_{i=1}^{p-1} |c_i|$ For a fixed number of overall bits v , e_0 and α is constant. The implementation error can thus be minimized by minimizing the partial sum at the internal nodes of an implementation tree. This sum is called the *cost of T* and is given by $C(T) = \sum_{i=1}^{p-1} |c_i|$ in [13].

Kao and Wang show in [13, Section 2] that the decision problem $C(T) \le K$, where $C(T)$ is the cost of the implementation tree T and K is a positive integer, is NP-hard. Using that result we have the following theorem.

THEOREM 1. *The Minimum-Error Fixed-point Set Range Sum (MEFxRS) problem is NP-hard.*

8. RELATED WORK

The range analysis problem has been studied extensively in the context of optimum bit-width allocation to intermediate variables in a fixed-point program, mostly in the DSP domain. Both static [18, 17, 25] and simulation-based [5, 20] approaches have been used. Existing work applied genetic programming as a technique to construct asynchronous circuits [23] or programs that can be model checked [14].

Jha [12] gives an algorithm for optimal fixed-point program synthesis based on inductive synthesis. His objective is to find the best fixed-point implementation for a given expression, and does not consider rewriting of expressions. Moreover, it takes several minutes to synthesize a fixed-point program corresponding to an expression, whereas our technique can synthesize a fixed-point program corresponding to an expression in seconds.

To our knowledge, the only work that considers rewriting of expressions to improve precision in the context of abstract interpretation is [11]. The authors develop an abstract domain for representing an under-approximation of mathematically equivalent expressions. They then use a local, greedy search to extract some expression with a more precise formulation in a floating-point implementation. While the computation of precision is similar to ours, this approach excludes many possible expressions due to the local search and due to the under approximation in the abstract domain. Also, no validation of the results obtained with a static analysis tool as compared to "true" errors is performed and thus the issue of imprecision when dealing with nonlinear expressions is missed. In previous work [21] the authors also considered fixed-point arithmetic. They used, however, only the maximum number of bits required to hold the integer part as the precision measure, which is too imprecise to distinguish many expressions.

9. CONCLUSION

We have presented an optimal fixed-point program synthesis methodology based on expression rewriting and genetic programming. We have shown how our technique helps improve the quality of controller implementations. Our controller synthesis tool can be seamlessly added to any automatic code generation tool flow to enhance its capability to generate correct-by-construction high performance controller software. Though we have presented our results on the benchmarks from the control engineering domain, our method is a general one, and can be used in enhancing the quality of a fixed-point implementation in any domain.

Abstract interpretation based technology has been very efficient in estimating the error bound of a fixed-point implementation of a linear controller program, and thus has been very useful in controller synthesis based on the minimization of the effect of the implementation error on the performance of the controller. However, the technique does not work very well for the nonlinear controllers. Our effort on estimating the error bound for a jet engine controller [16] reveals that the abstract interpretation based error estimation technique may end up providing very pessimistic bounds. This benchmark motivates research towards enhancing the capabilities of abstract interpretation based tools to deal with nonlinear arithmetic.

10. REFERENCES

[1] T. Aamodt and P. Chow. Embedded ISA Support for Enhanced Floating-Point to Fixed-Point ANSI C Compilation. In *CASES*, 2000.

[2] A. Anta, R. Majumdar, I. Saha, and P. Tabuada. Automatic verification of control system implementations. In *Proceedings of EMSOFT*, 2010.

[3] A. Anta and P. Tabuada. To Sample or not to Sample: Self-Triggered Control for Nonlinear Systems. *IEEE Trans. Automatic Control*, 55(9), 2010.

[4] K. J. Astrom and R. M. Murray. *Feedback systems.* Princeton University Press, 2008.

[5] P. Belanovic and M. Rupp. Automated Floating-point to Fixed-point Conversion with the Fixify Environment. In *Proc. Rapid System Prototyping*, 2005.

[6] E. Darulova and V. Kuncak. Trustworthy numerical computation in Scala. In *OOPSLA*, pages 325–344, 2011.

[7] S. Ebbesen, P. Kiwitz, and L. Guzzella. A generic particle swarm optimization function for Matlab. In *American Control Conference*, 2012.

[8] C. F. Fang, R. A. Rutenbar, and T. Chen. Fast, Accurate Static Analysis for Fixed-Point Finite-Precision Effects in DSP Designs. In *ICCAD*, 2003.

[9] M. Green and D. J. N. Limebeer. *Linear robust control.* Prentice Hall, August 1994.

[10] J. P. Hespanha. *Linear systems theory.* Princeton University Press, 2009.

[11] A. Ioualalen and M. Martel. A New Abstract Domain for the Representation of Mathematically Equivalent Expressions. In *SAS*, 2012.

[12] S. Jha. *Towards Automated System Synthesis Using SCIDUCTION.* PhD thesis, University of California at Berkeley, 2011.

[13] M.-Y. Kao and J. Wang. Efficient minimization of numerical summation errors. In *Automata, Languages and Programming.* Springer, 1998.

[14] G. Katz and D. Peled. Model Checking-Based Genetic Programming with an Application to Mutual Exclusion. In *TACAS*, pages 141–156, 2008.

[15] J. Kennedy and R. Eberhart. Particle swarm optimization. *In Proceedings of IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[16] M. Krstic and P. Kokotovic. Lean backstepping design for a jet engine compressor model. In *Proceedings of IEEE Conf. Control App.*, pages 1047–1052, 1995.

[17] J. A. López, C. Carreras, and O. Nieto-Taladriz. Improved Interval-Based Characterization of Fixed-Point LTI Systems with Feedback Loops. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26, 2007.

[18] D. Lee, A. A. Gaffar, R. C. C. Cheung, O. Mencer, W. Luk, and G. A. Constantinides. Accuracy-Guaranteed Bit-Width Optimization. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 25(10), 2006.

[19] R. Majumdar, I. Saha, and M. Zamani. Synthesis of minimal-error control software. In *EMSOFT*, pages 123–132, 2012.

[20] A. Mallik, D. Sinha, P. Banerjee, and H. Zhou. Low-Power Optimization by Smart Bit-Width Allocation in a SystemC-Based ASIC Design Environment. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 26(3), 2007.

[21] M. Martel. Enhancing the implementation of mathematical formulas for fixed-point and floating-point arithmetics. *Formal Methods in System Design*, 35(3), 2009.

[22] The MathWorks^{TM}. Simulink Fixed Point. `http://www.mathworks.com/products/simfixed/`.

[23] T. McConaghy, P. Palmers, M. Steyaert, and G. G. E. Gielen. Trustworthy Genetic Programming-Based Synthesis of Analog Circuit Topologies Using Hierarchical Domain-Specific Building Blocks. *IEEE Trans. Evolutionary Computation*, 15(4):557–570, 2011.

[24] P. McLane, L. Peppard, and K. Sundareswaran. Decentralized feedback controls for the brakeless operation of multilocomotive powered trains. *IEEE Trans. Autom. Control*, 21(3), 1976.

[25] W. G. Osborne, R. C. C. Cheung, J. G. F. Coutinho, W. Luk, and O. Mencer. Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems. In *Proc. FPL*, pages 617–620, 2007.

[26] R. Poli, W. B. Langdon, and N. F. McPhee. *A Field Guide to Genetic Programming.* Lulu Enterprises, 2008.

[27] H. H. Rosenbrock. *Computer-Aided Control System Design.* Academic Press, 1974.

[28] Control Tutorial for Matlab and Simulink. Available online at http://www.library.cmu.edu/ctms/ctms/.