

The Evolutionary Design and Synthesis of Non-Linear Digital VLSI Systems

Robert Thomson and Tughrul Arslan
Department of Electronics and Electrical Engineering
The University of Edinburgh
Scotland
{Robert.Thomson,Tughrul.Arslan}@ee.ed.ac.uk

Abstract

This paper describes a multi-objective Evolutionary Algorithm (EA) system for the synthesis of efficient non-linear VLSI circuit modules. The EA takes the specification for a non-linear block, and converts it into a technology independent netlist, specified in the Verilog hardware description language. The hardware designs are based upon high-level components such as adders and multipliers. The circuit designs that are produced are near-optimal with respect to silicon area and longest-path delay.

The performance of the EA is enhanced through the use of local searches. These searches are embedded within the genetic operators, and enable the rapid evaluation of large numbers of designs. The use of searches increases the power of the EA system, without forfeiting the benefits of using a population of solutions.

The system is demonstrated with several test problems. Results are presented for the discovery of correct designs, and also regarding the quality of the evolved designs. The most complex designs have areas as large as $200,000\mu\text{m}^2$ in a $0.18\mu\text{m}$ technology.

1. Introduction

This paper describes the use of Evolutionary Algorithms for the creation of non-linear circuits. The designs that are evolved are based around high-level components such as adders, subtractors and multipliers. The input to the system is a high-level behavioural description of a non-linear device, and the output is a set of near-optimal circuit netlists. The EA attempts to produce designs that are both functionally correct, and also efficient in terms of area and longest-path delay.

Evolutionary Algorithms are a class of stochastic optimisation methods that were inspired by natural evolution [7]. They iteratively improve the quality of a population of solutions, through the application of random modifications to

the solutions, and by duplicating the best solutions while eliminating the worst solutions. Optimisation is done with respect to some application-specific measure of solution fitness. EAs are useful for discovering near-optimal solutions to highly complex problems.

EAs have been applied to a wide variety of circuit synthesis problems. This includes the designs of analogue hardware and gate-level digital hardware. Of most interest here is the design of functional-level digital hardware [9]. This is hardware that is based around high-level components such as adders and subtractors.

1.1 Non-Linear Filtering

Although a large amount of signal processing is linear, there are many cases where a signal transformation cannot be accurately expressed using linear methods, and non-linear techniques are necessary. Typical non-linear filter applications include the correction of non-linear distortion introduced either by sensors, communication channels or signal processing, and compensation for non-linearity in output devices. Non-linear mathematical functions, such as trigonometric, logarithmic and hyperbolic functions, also have a wide variety of applications.

The system has currently been demonstrated with a class of non-linear circuits known as Volterra filters [16]. A discrete Volterra filter can be defined by a polynomial in terms of its inputs. Volterra filters can be classified according to the highest-order terms that they compute. First, second, and third-order Volterra filters are known as linear, quadratic, and cubic filters respectively. In general, higher order filters tend to offer the highest accuracy. The computational complexity of Volterra filters tends to grow very rapidly as the filter order is increased, so in many cases only the low-order terms of a response are implemented.

The EA system described in this paper can produce non-linear circuits that have one or more inputs, and one or more outputs. These devices can be described using a set of polynomials based on the inputs. The EA system can also pro-

duce linear designs, as they are a subclass of Volterra filters.

1.2. Filter Realisation

The creation of resource-efficient filter designs is a complex problem. Typically, the most obvious designs are highly sub-optimal. More efficient designs can be created through the elimination of unnecessary components, the sharing of components between different functions, and the use of resource-efficient components. In many cases the most optimal design is not obvious. In fact, the computational complexity of these optimisation problems can be so severe, that the only reasonable approach is to attempt to discover near-optimal solutions using heuristic algorithms.

Primitive Operator Filters (POFs) [1] are filters that avoid the use of multipliers, and instead use *Primitive Operators* such as addition, subtraction, negation, and bit-shifting. These operations are low-cost, where cost is measured in terms of delay, power or silicon area.

The simplest Primitive Operator design problem is the creation of a circuit that multiplies by a constant value. Constant multipliers are very commonly used as components in larger designs. There are a number of ways in which a constant multiplier can be designed — the Canonic Signed Digit (CSD) [10] method is one of the best known, although other methods [4, 5] can give better results.

If one input is multiplied by more than one constant, further efficiency savings can be achieved by sharing hardware between the multiplications. Unfortunately, the task of designing such a multiplication block is extremely complex. There are a number of heuristic methods that give near-optimal results [1, 6].

Several primitive operator multiplication blocks can be combined into a block that performs linear transforms, again increasing the efficiency of the design. Much of the work in this area relates to the creation of optimised implementations of common transforms such as the Fourier Transform or the Discrete Cosine Transform [12] — much of this work is application-specific and cannot be extended to other transforms. Our previous work dealt with the use of EAs for the creation of linear transform circuits [18].

The methods mentioned so far have been linear. It is possible to extend the above concepts for applications that require non-linearity, through the use of non-linear components together with the linear primitive operator components. In this paper, we investigate the use of multipliers as non-linear components. Schemes that combine other types of non-linear component together with linear primitive operator parts are also possible.

Normally, Volterra filter designs are mathematically derived from the Volterra kernel — see [17] for an example of how this can be done. The technique described in this paper allows for greater hardware efficiency. This is possible be-

cause there is more potential for the sharing of components, and also because hardware-efficient approximations to the filter coefficients can be chosen. Although hardware savings are possible, the realisation of efficient hardware is a computationally complex task.

1.3. Graph Chromosomes

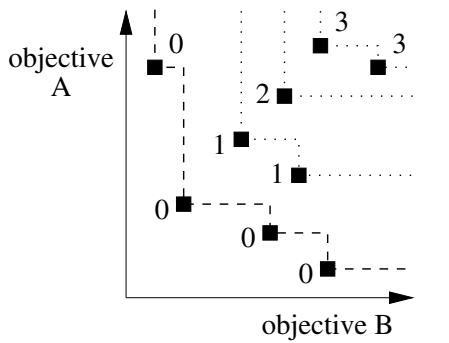
Many EAs represent each chromosome using a bit-string or a series of values. This type of representation is useful for many problems as it is simple, and amenable to the application of mutation and crossover operators. However, linear chromosome representations do tend to be a poor representation for electronic circuit topology. It is difficult to devise a linear representation for a circuit design, such that similar circuits have similar chromosomes. Linear representations tend to be prone to epistasis — a small change to the chromosome can produce a major change in the properties of the phenotype. There are two main approaches to solving this problem; improvements can either be made to the system that converts from the genotype to the phenotype, or else the genotype can be redesigned so that it is more similar to the phenotype. Enzyme Genetic Programming [13] is an example of the former approach, while methods such as Cartesian Genetic Programming and the use of graph chromosomes are examples of the latter approach.

Cartesian Genetic Programming (Cartesian GP) [15, 14] is the use of a 2-dimensional planar chromosome, rather than a 1-dimensional linear chromosome. Genetic Programming (GP) [11] represents a chromosome with a tree. Cellular Encoding [8] is a method for converting a tree into a graph, which can be used together with Genetic Programming for the evolution of digital circuit designs. Lastly, graph-based techniques [2] use the most direct mapping between a genotype and a phenotype; typically each part of a graph chromosome corresponds directly to a part of a circuit design. The simplicity of the genotype-phenotype mapping is the reason that graph chromosomes have been used by the system described in this paper.

1.4. Multi-Objective EAs

Single-objective EAs attempt to optimise with respect to a single fitness measure. This is insufficient for many real-world problems, which can have multiple, conflicting objectives. This resulted in the development of multi-objective EAs [3]; EAs that have multiple fitness measures.

The most significant difference between single-objective EAs and multi-objective EAs is that a multi-objective EA can produce more than one ‘best’ design. The output from a multi-objective EA is a set of *non-dominated* solutions. Each non-dominated solution makes a different compromise between conflicting objectives. The non-dominated



key: ■ Individual
 --- Non-dominated individuals
 Dominated ranks

Figure 1. Ranking of solutions in a multi-objective EA. The individuals are labelled with the ranks assigned by the non-dominated sorting technique.

solutions are all ‘best’ solutions, but they have different qualities. This is illustrated in figure 1.

Population diversity is especially important for multi-objective EAs. If the population is homogeneous, the solution set will not include a wide variety of solutions. There are two ways that a diverse population can be encouraged. Firstly, the fitness measures can be combined using a method that does not favour any particular combination of fitness values. Secondly, diversity can be explicitly encouraged through mechanisms such as niching [3], which reward rare solutions.

1.5. Paper Contents

The rest of this paper is structured as follows. Section 2 describes the EA system. Section 3 includes results for various test problems. Section 3 is split into three, with subsection 3.1 describing the application of the system to a simple problem, subsection 3.2 describing the application of the EA to some other problems, while subsection 3.3 investigates a limitation of the current system and how this limitation can be overcome. The conclusions are in section 4.

2. Description of the EA System

2.1. Problem Representation

A Volterra filter can be represented in the following form:

$$y(n) = \sum_{k_1} h_1(k_1)x(n - k_1)$$

Table 1. The mapping between graph nodes and hardware components.

Operation	Components used
$x + y$	adder
$x - y$	subtractor
$-(x + y)$	adder, negator
xy	multiplier
$-xy$	multiplier, negator

$$+ \sum_{k_1} \sum_{k_2} h_2(k_1, k_2)x(n - k_1)x(n - k_2)$$

$$\vdots$$

$$+ \sum_{k_1} \cdots \sum_{k_M} h_M(k_1, \dots, k_M)x(n - k_1) \cdots x(n - k_M)$$

where the symbols are:

$h_p(k_1, \dots, k_p)$ p th-order Volterra kernel
 $y(\cdot)$ output values
 $x(\cdot)$ input values

The Volterra kernel values are the set of constant coefficients that define the filter.

The EA system can produce circuits that have several outputs. In that case, each output can be described as above. There is no need to limit this characterisation to the output values; all of the intermediate values in a design can also be described in a similar fashion.

2.2. The Chromosome

The chromosome has been encoded as an acyclic directed graph. The nodes in the graph represent components, while the edges represent the interconnections between components. This encoding means that operations on the chromosome correspond to similar operations on the phenotype.

Each node in a graph represents a two-input operation. The basic operation can either be addition or multiplication. The inputs to a node can be shifted or negated. In other words, the inputs to each node can be multiplied by $\pm 2^x$, for $x \in \mathbb{Z}$. The node then adds or multiplies these scaled input values.

The chromosome can be converted into a hardware netlist with very little effort. Each node in the graph corresponds to one or two components in the final design. The mapping is described by table 1. The current system for hardware implementation makes use of fixed-point components. All components are the same width. The multipliers that are used are slightly differently from normal integer

multipliers in that they behave as if all values are between -1 and 1; in other words the inputs and output are in a fixed-point representation which does not have an integer part. This representation prevents overflows from happening, although the precision is still limited by the size of the values used.

All of the functional simulations are carried out using floating point values. When the netlists are produced, the system will shift all of the values such that overflows cannot happen, and so that the maximum number of bits of accuracy are used. In effect, the use of shifts gives each value a fixed exponent; the shift scales the value to the magnitude that is most useful. The alteration of shift values takes account of the shift values in the chromosome, so that the function of the design is not altered.

2.3. Objectives

The EA has three objectives: the design should function correctly, it should contain as few components as possible, and it should have the shortest possible critical path. These objectives can conflict with each other, so each design is a compromise between the objectives.

The functionality of a design is measured using two values. The first value counts the number of terms that are present and which should be present. The second value measures the error between the desired polynomial and the actual polynomial. For designs that have multiple outputs, the values are summed for all outputs. The first of these values should be maximised, while the second should be minimised. The first value takes precedence over the second. We could describe the relationship between two functional fitness values as follows:

$$\begin{aligned} (t_1, e_1) \succ (t_2, e_2) & \text{ if } t_1 > t_2, \\ (t_1, e_1) \succ (t_2, e_2) & \text{ if } t_1 = t_2, e_1 < e_2, \end{aligned}$$

where \succ denotes 'is better than'.

As an example of how the functionality score is calculated, consider how the response $3i_1^2 + 3i_1 + 7i_2$ compares to the specification $6i_1i_2 + 5i_1 + 8i_2$. First of all there are i_1 and i_2 terms present in both polynomials, so the first part of the score is 2, denoting that there are two desired terms. The second value is the sum of squares difference between the terms in the polynomials. This can be illustrated as follows:

$$\begin{array}{r} \text{wanted: } 6i_1i_2 + 0 + 5i_1 + 8i_2 \\ \text{actual: } 0 + 3i_1^2 + 3i_1 + 7i_2 \\ \text{terms: } 0 + 0 + 1 + 1 = 2 \\ \text{error: } 6^2 + 3^2 + 2^2 + 1^2 = 50 \end{array}$$

Therefore, the final score is (2, 50).

The reason for this two-tier functional fitness system is that it very strongly rewards designs that have all of the desired terms. If only the error is used as the functional fitness

Table 2. Component properties.

Component	Area (μm^2)	Delay (ns)
adder	1016	2.14
subtractor	1537	1.68
negator	870	0.99
multiplier	32635	7.97

measure, the resulting designs often omit many of the specified terms.

If the goal is to create completely functionally perfect designs, the EA will continually reduce the functional error by adding components. The result is circuit designs which are very close to the functional specification, but which use an unreasonably large number of components. A more reasonable approach is to specify a level of functional error that is acceptable, and to consider all of the designs that surpass this level to be equally good. For this reason, the EA system lets the user specify an acceptable error value. We describe the designs that reach this level as 'correct'.

The area and delay for each design are estimated using the values shown in table 2. These figures were derived from a library of $0.18\mu\text{m}$ components. The figures relate to 16-bit components. Note that multipliers are far more expensive than other components, both in terms of delay, and especially with respect to area.

The area of a complete design is estimated by summing the areas of all of the components. The longest-path delay is estimated by finding the path through the design that has the largest sum of delay values.

This hardware modelling system gives approximate results. The models not take account of the effects of wiring on the properties of a design. The delay model also ignores the fact that wires in a bus can have different delays, therefore it tends to overestimate the overall delay. A major strength of the modelling system is its speed.

It is worth noting that the values produced by the hardware models do not need to be accurate, they just need to preserve the ordering of designs. For example, if one design is slower than another in reality, this should be reflected by the model, even if the values that the model returns are otherwise inaccurate. The EA only needs to know whether it should prefer one design over another; additional knowledge is unnecessary.

2.4. The Evolutionary Algorithm

One of the main problems when evolving digital circuits, is the fact that most of the modifications that are made to a chromosome are functionally destructive. For this reason, a $(\mu+\lambda)$ system has been used together with elitism, ensuring that the best individuals are not eliminated.

An initial population of 100 individuals is created. The initial population is expanded through the use of size-2 tournament selection and the mutation operators. The expanded population therefore consists of 200 chromosomes: 100 parents and 100 mutant children. A new population is created by preserving 100 individuals from the expanded population. Elitism preserves the non-dominated individuals — if there are more than ten non-dominated individuals, then only the ten most functional are preserved. The rest of the new population is chosen using size-2 tournament selection.

There are three fitness measures — functional correctness, area and delay. The fitness measures are combined using the non-dominated sorting method [7, 3]. This assigns each individual a rank relative to the rest of the population. The best, non-dominated individuals are assigned a rank of zero, while lower-quality individuals are assigned higher numbers. An example of the ranking performed by this scheme is illustrated in figure 1. This ranking technique tends not to be biased towards any particular combination of fitness measures, and when combined with niching, will produce a diverse set of results. For a full description of this technique, see Deb [3].

The ranking scheme described above is slightly complicated by the need to encourage population diversity, and by the need to discourage non-functioning designs. Diversity is encouraged through the use of niche counts [3]. In this EA system, the niche count is the number of individuals that have a particular set of values for the three objectives. Thus if an individual has a high niche count, it means that there are many other equivalent individuals in the population. If the two individuals in a tournament have the same rank, the individual with the lowest niche count wins the tournament. This encourages diversity by encouraging the selection of rare individuals.

Non-functioning designs must be discouraged, because they can easily dominate the population. The problem is that many designs function poorly, but have a low area and longest-path latency, resulting in a high rank. However, individuals can have a less than perfect functional performance and still be useful, both as individuals and also as parents. The approach that was taken was to label the least functional individuals as ‘broken’. If a tournament contains one individual that is broken, and another that is not, the unbroken individual wins, regardless of rank or niche count. Individuals are labelled as broken if they have more than 100 times the functional error, or less than a quarter as many of the desired terms, of the most functional individual in the population.

2.5. Genetic Operators

All of the genetic operators are mutational. The possibility of adding a crossover operator, such as the one used in

Table 3. The genetic operators.

Operator	Probability	Search?
Connection modification	1/6	yes
Adder/subtractor insertion	1/6	yes
Multiplier insertion	1/6	yes
Shift modification	1/6	yes
Component removal	1/6	no
Component reorganisation	1/6	no

the EGG system [2], has not been investigated. The full set of genetic operators is listed in table 3. One of these operators is applied immediately after each new child individual is created.

Some of the genetic operators perform local searches. The searches serve two purposes. Firstly, the searches reduce the computational cost of investigating each design. This is possible because the searches investigate many similar designs, so duplicated computations can be avoided. Secondly, most mutations are functionally destructive. The EA will often mutate an individual and then later eliminate the mutant because it functions very poorly. The local searches ensure that non-functional designs are immediately eliminated, reducing the amount of effort spent processing useless designs.

The searching genetic operators evaluate the function of a series of modified designs. An edge is chosen at random, and a series of modifications are made to that edge. The modification that results in the most functional design is chosen as the result of the mutation.

Before a search is performed, every node in the design is labelled with its response, which is a polynomial based upon the inputs. As all of the modifications change the value of one edge, the value on that edge is also considered to be a variable in these polynomials. Thus outputs that depend on the chosen edge will have an unknown value in their polynomials. When a search is performed, various different edge responses are substituted in place of this unknown, and the resulting output responses are evaluated.

An example of node labelling is shown in figure 2. In this case, the nodes are labelled with polynomials in terms of inputs a and b . The edge at the centre of the figure is chosen for modification, and it is labelled with the variable x . Therefore, the nodes that depend on the chosen edge have an x in their responses. The response of the unmodified system can be found by taking the output response, $abx + b$, and using the substitution $x = a + b$ to get $a^2b + ab^2 + b$. The effects of assigning different values to x can also be simulated. Changing a connection could lead to x taking the values $x = a$, $x = b$, or $x = ab$. Inserting a shift or negation could result in x taking values like $x = \pm 2^k(a+b)$, where $k \in \mathbb{Z}$. Finally, inserting a new node can result in the

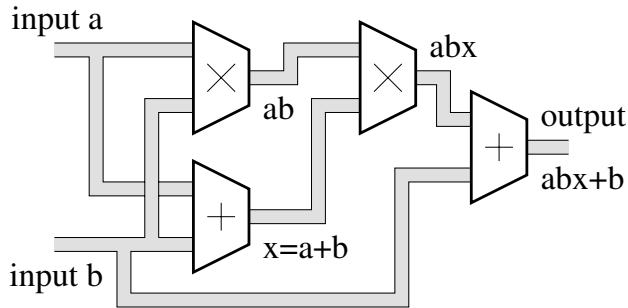


Figure 2. An example of how node responses are characterised.

creation of new values such as $x = (ab) + (a)$ or $x = (b)(a + b) = ab + b^2$.

There are four genetic operators that use local searches. The ‘connection modification’ operator tries connecting the edge to different nodes, also randomly applying shifts and negations. The ‘adder/subtractor insertion’ operator is very similar, except it adds a new value on to the existing edge, resulting in the insertion of a new adder or subtracter node. The ‘multiplier insertion’ operator considers the products of pairs of nodes, which can be randomly shifted and negated. The ‘multiplier insertion’ operator leads to the creation of a new multiplier node. Lastly, the ‘shift modification’ operator does not attempt to connect the edge to a new source, but instead tries applying a range of different shift and negation combinations to the edge, effectively scaling the value on the edge.

With reference to figure 2, the searching genetic operators could consider the following substitutions. The ‘connection modification’ operator could try changes such that $x = ab$ or, with shifts and negations, perhaps $x = -2^3(b)$. Similarly, the ‘adder/subtractor insertion’ operator could try $x = (a + b) + (ab)$ or $x = (a + b) + (-2^3(b))$. The ‘multiplier insertion’ operator might consider $x = (b)(b) = b^2$ or $x = -2^2(ab)(a)$. The shift modification operator considers values of the form $x = \pm 2^k(a + b)$, with $k \in \mathbb{Z}$.

The local searches only take account of functionality, and ignore area and longest-path delay. The EA *does* select according to those properties, so the algorithm as a whole will attempt to optimise with respect to all of the objectives.

When performing a search, the pre-existing value is always entered into the search. If it is chosen, the chromosome is not modified. This limits the destructiveness of the genetic operators, but might inhibit exploration. If a more explorative EA is required, the genetic operators could be altered so that they always modify the chromosome.

The size of the searches can be altered. Ideally, it should be large enough that approximately one useful modifica-

tion is discovered per search. When testing the EA system, searches of size 20 were used. The ‘shift modification’ operator considered all of the shifts of between 8 bits left and 8 bits right, both negated and unnegated.

The component removal operator removes a single node from the design. One of the inputs to the removed node is eliminated, while the other input replaces the output value from the removed node. In effect, a two-input node is replaced with an edge.

The ‘component reorganisation’ operator attempts to reduce the longest path delay for a design. To do this it selects two connected nodes which are of the same type; both adders or else both multipliers. It then uses associativity, replacing $(a + b) + c$ with $a + (b + c)$, or replacing $(a.b).c$ with $a.(b.c)$. If this operator is applied to nodes that are on the critical path, it can sometimes reduce the length of the critical path by one component. This operator also helps to increase genotypic population diversity, because it shuffles the components in a design, without changing the overall function of the design.

3. Results

3.1. An Example Problem — the Sine Function

The Taylor series approximation of a sine function can be written as follows:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

This approximation gets increasingly inaccurate as x is moved away from 0, however if x is limited to the range $-\frac{\pi}{2} \leq x \leq \frac{\pi}{2}$, three terms are sufficient for results that are accurate to within 1%. If five terms are used, the results will have the equivalent of more than 16 bits of accuracy.

The EA was applied to the following sine approximation:

$$\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!}$$

Twenty runs of 100 generations were performed, and the maximum desired error was set to 0.001. 17 of the 20 runs rapidly produced designs that met the specification, while the other three runs produced designs that only approximated the first term correctly. The failure to produce high-order terms is due to the lack of a squared term in the problem specification, which discourages the insertion of multipliers into linear circuits. The functional error for the functionally best population members is shown in figure 3.

The quality of the designs that are produced is highly constrained due to the fact that the designs must contain three multipliers, and the three multipliers must be on the

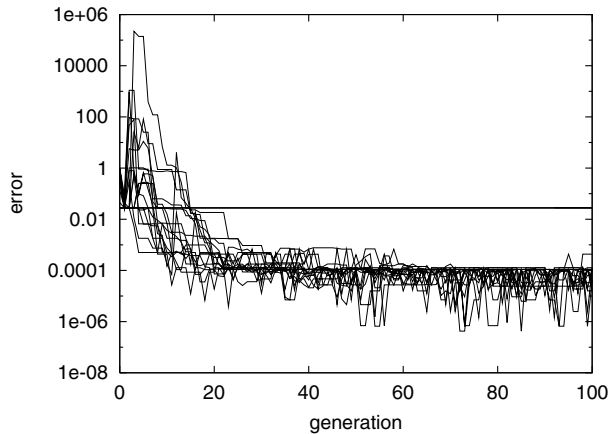


Figure 3. Functional errors for the most functional individuals, for 20 runs of the sine problem. Error values of 0.001 or below are desired.

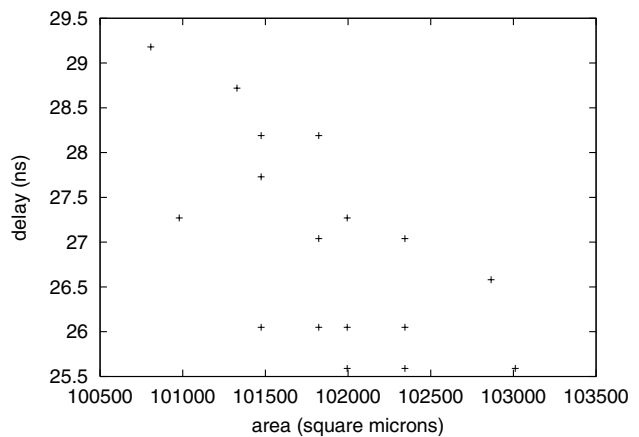


Figure 4. Properties of the sine circuits that meet the specification.

longest path. This is something that is inherent to this particular problem. This constraining of the solutions is apparent in figure 4, which shows the properties of the correct, non-dominated results from each of the 20 EA runs.

Of the results shown in figure 4, the lowest area and lowest delay designs are particularly interesting. The lowest area design has the following response:

$$\sin(x) \approx x - 0.15625x^3 + 0.00390625x^5$$

This is performed using only three multipliers, two adders and a negator. The automatically produced Verilog for this circuit is shown in netlist 1. The results of simulating this design are shown in figure 5. This circuit works between $-\frac{\pi}{2}$ and $\frac{\pi}{2}$, and the simulation results show it is accurate to within 1% throughout that range.

The lowest-delay design has three multipliers and one subtracter on the critical path. This seems to be the minimum possible delay for these components and this specification. Note that a series of three multipliers is needed in order to compute x^5 , and at least one extra component is then needed in order to produce the complete response. With the component values used here, a subtracter is faster than an adder.

3.2. Application to Larger Problems

The system was tested on polynomials that have random coefficients. Two types of polynomial were used. The first set of polynomials are factorisable. They are of the form:

$$(k_1i_1 + k_2i_2)(k_3i_1 + k_4)(k_5i_2 + k_6)$$

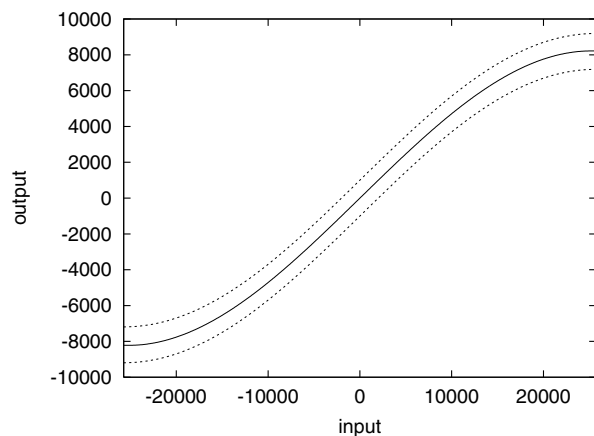


Figure 5. Results of simulating a Verilog implementation of the sine circuit (central line), compared with perfect sines (top and bottom lines).

```

module nonlinear_netlist (i0,o0);
input [15:0] i0;
output [15:0] o0;
wire [15:0] w0,w1,w2,t0,w3,t1,w4,t2;
adder_16bit m0 (w0, {{1{i0[15]}}},i0[15:1]},
               {{3{w4[15]}}},w4[15:3]});
adder_16bit m1 (w1, w0,
               {{4{w3[15]}}},w3[15:4]});
mult_16bit m2 (t0, i0, i0);
negater_16bit m3 (w2, t0);
mult_16bit m4 (t1, w2, w0);
assign w3 = {t1[14:0],1'b0};
mult_16bit m5 (t2, i0, w2);
assign w4 = {t2[14:0],1'b0};
assign o0 = w1;
endmodule

```

Netlist 1: The netlist for the most area-efficient evolved sine circuit.

where $k_1 \dots k_6$ are random real numbers between -10 and 10 , and i_1 and i_2 are the inputs. These polynomials were used for the one output and two output cases. These test problems are called ‘factored 1’ and ‘factored 2’. The test specifications are respectively:

$$o_1 = -48.38i_1^2i_2 + 219.2i_1i_2 - 44.85i_1i_2^2 + 99.24i_2^2 + 120.9i_1^2 - 267.6i_1 + -248.1i_2$$

and:

$$o_1 = 62.48i_1^2i_2 + 146.3i_1i_2 + 46.46i_1i_2^2 + 94.61i_2^2 + 25.62i_1^2 + 52.16i_1 + 38.79i_2$$

$$o_2 = -109i_1^2i_2 + 9.818i_1i_2 - 43.37i_1i_2^2 - 22.32i_2^2 + 165.8i_1^2 + 85.32i_1 + 33.94i_2$$

The system was also tested on larger polynomials, which are not factorisable. These polynomials include all of the possible first and second-order terms for a set of inputs. Each term is multiplied by an integer between -100 and 100 . The first of these problems has three inputs and three outputs, and we have called it ‘random 3×3 ’. It is specified as follows:

$$o_1 = 13i_1 - 44i_1^2 - 26i_2 - 16i_2i_1 - 36i_2^2 - 35i_3 + 63i_3i_1 + 70i_3i_2 - 86i_3^2$$

$$o_2 = 98i_1 - 76i_1^2 - 24i_2 - 98i_2i_1 + 16i_2^2 + 73i_3 + 100i_3i_1 - 13i_3i_2 - 36i_3^2$$

$$o_3 = 91i_1 - 7i_1^2 + 67i_2 + 14i_2i_1 - 90i_2^2 + 83i_3 - 10i_3i_1 - 48i_3i_2 + 91i_3^2$$

The last problem has four inputs and four outputs, and is called ‘random 4×4 ’:

$$o_1 = 43i_1 + 21i_1^2 - 89i_2 - 81i_2i_1 + 83i_2^2 + 26i_3 + 95i_3i_1 + 62i_3i_2 + 12i_3^2 - 17i_4 - 82i_4i_1 + 58i_4i_2 + 87i_4i_3 - 74i_4^2$$

$$o_2 = -82i_1 - 61i_1^2 + 35i_2 + 8i_2i_1 - 71i_2^2 + 90i_3 - 43i_3i_1 - 25i_3i_2 - 2i_3^2 - 91i_4 + 24i_4i_1 - 92i_4i_2 - 22i_4i_3 - 39i_4^2$$

$$o_3 = 79i_1 + 20i_1^2 + 15i_2 + 41i_2i_1 - 16i_2^2 + 85i_3 + 87i_3i_1 + 43i_3i_2 - 39i_3^2 - 74i_4 - 69i_4i_1 - 45i_4i_2 + 9i_4i_3 - 4i_4^2$$

$$o_4 = 15i_1 - 69i_1^2 + 52i_2 + 20i_2i_1 - 14i_2^2 + 71i_3 - 99i_3i_1 -$$

Table 4. Test problems.

Problem	Inputs	Outputs	Order	Terms
sine	1	1	5	3
factored 1	2	1	3	7
factored 2	2	2	3	14
random 3×3	3	3	2	27
random 4×4	4	4	2	56

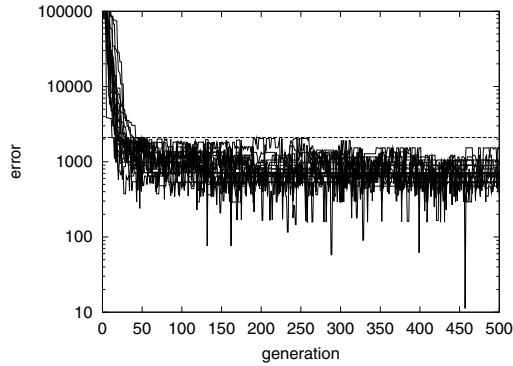
$78i_3i_2 - 35i_3^2 + 49i_4 + 29i_4i_1 - 60i_4i_2 + 65i_4i_3 - 5i_4^2$
The properties of these problems are listed in table 4.

The EA was used on the above test problems. For each of these problems, the maximum acceptable error value was set to one hundredth of the sum of the squared coefficients. Twenty runs were attempted for each problem. The run lengths were 500 generations for the two factorised problems, 1000 generations for the ‘random 3×3 ’ problem, and 2000 generations for the ‘random 4×4 ’ problem.

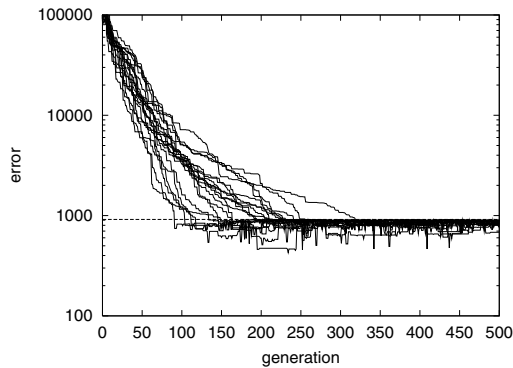
The evolutionary progress with three of the problems is illustrated in figure 6. The EA was always able to find correct solutions for the simplest three problems. The ‘random 4×4 ’ problem was not completed, for reasons that are discussed below.

Figure 7 shows the properties of the correct designs. The complexity of this problem domain prevents the determination of whether a design is truly optimal in all but the simplest cases. For this reason, we will only consider the multiplications when assessing the optimality of the results. The EA can solve the ‘factored 1’ problem using only two multiplications, which is the minimum number of multiplications for a cubic filter. The best solutions to the ‘factored 2’ problem have only two multipliers on the critical path, which is the minimum number. The lowest-area solutions to the ‘factored 2’ problem use three multipliers. As each output requires a minimum of two multiplications, this shows that hardware is being shared between the two outputs. The smallest solution to the ‘random 3×3 ’ problem also uses three multipliers. The fastest evolved solutions to the ‘random 3×3 ’ problem do not have the lowest possible critical path length. For example, consider a hand-made solution that generates each term independently, using two multipliers in series, and then sums the terms using a four-deep tree of adders. Such a solution would have a longest-path delay which is at least 10% shorter than the fastest evolved solution, while its area would be far worse. Therefore, the evolved solution does not have a minimal longest-path delay, although it could still be Pareto-optimal.

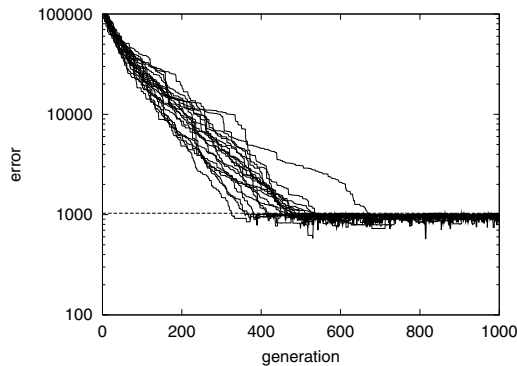
Figure 8 illustrates the complexity of the larger designs. It was rendered by the BuildGates synthesis system. The boxes in figure 8 represent adders, subtractors, negators and multipliers.



(a) Factored 1.



(b) Factored 2.



(c) Random 3×3 .

Figure 6. The lowest error values, by generation, for 20 runs of three different problems. The dotted lines show the target error values.

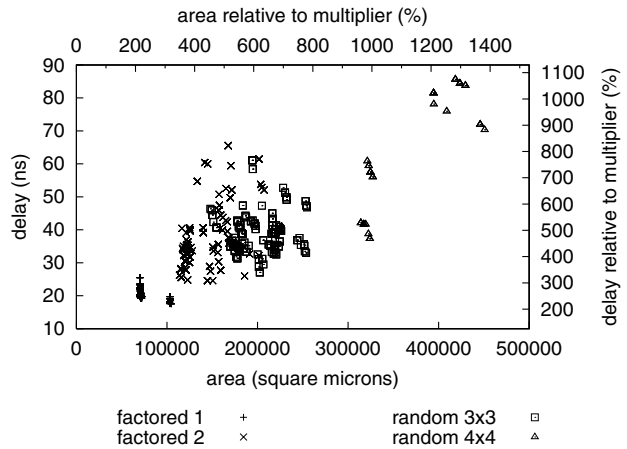


Figure 7. Properties of the correct solutions for four problems.

3.3. Scalability of the Current System

When attempting to complete the ‘random 4×4 ’ problem, the EA system took between 90 minutes and more than a day to process each run. The total time required for these runs was so large that the planned 20 runs were not completed. For comparison, the ‘factored 1’ problem took 1–2 minutes, the ‘factored 2’ problem took 5–10 minutes, and the ‘random 3×3 ’ problem took roughly 15 minutes, although these problems did require fewer generations. Although it ran slower than for the other problems, the system was able to evolve correct designs for the ‘random 4×4 ’ problem.

The reason that the evaluation of individuals takes a longer time with the ‘random 4×4 ’ problem is that the characterisation is very complex. Many of the polynomials that are used to describe nodes include large numbers of undesired terms. These extra terms generally have a very small magnitude, so they do not necessarily prevent an individual from meeting the functional specification. The extra terms cause an explosion in the complexity of operations such as polynomial multiplication, and hence the entire system is slowed down.

There are two ways that this problem could be tackled. Either extra terms can be discouraged, or else the amount of computation can be reduced.

If unwanted terms were punished, the EA would concentrate on the designs that are easiest to evaluate. This would probably speed up the system, however it could prevent the discovery of some useful designs. An alternative version of this scheme would be to directly reward or punish individuals according to the length of time that they take to evaluate.

The computational cost of evaluating individuals could

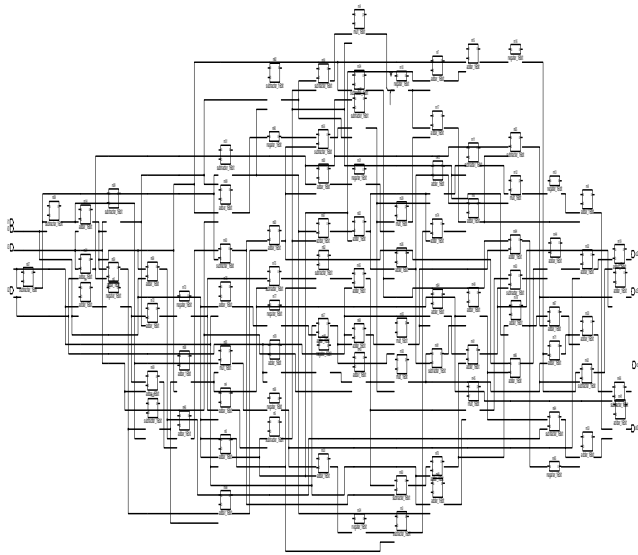


Figure 8. The fastest correct design evolved for the random 4×4 problem.

be reduced if the representation for a polynomial were changed. Rather than calculating all of the unwanted terms exactly, they could be approximated. For example, when creating a second-order design, the terms that are higher than second-order could be represented by a single worst-case error value.

4. Conclusions

This paper has described an Evolutionary Algorithm for the synthesis of non-linear digital circuits. The input to this system is a set of polynomials that describe the desired response, and the output from the system is a set of netlists that can be used for the implementation of the circuit. The netlists are written in the Verilog hardware description language. The designs that this system produces are near-optimal with respect to area and longest-path delay.

The EA makes use of a local searches, which are embedded within the genetic operators, and which increase the power of the system. These searches make use of characterisation information that is calculated for each point in the design.

The EA system was demonstrated on a set of problems, and the results were analysed with respect to search performance, as well as the quality of the generated designs.

References

[1] D. R. Bull and D. H. Horrocks. Primitive operator digital filters. *IEE Proceedings — Circuits, Devices and Systems*,

138(3):401–412, June 1991.

[2] D. Chen, T. Aoki, N. Homma, T. Terasaki, and T. Higuchi. Graph-based evolutionary design of arithmetic circuits. *IEEE Transactions on Evolutionary Computation*, 6(1):86–100, Feb. 2002.

[3] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. John Wiley and Sons, 2001.

[4] A. G. Dempster and M. D. Macleod. Constant integer multiplication using minimum adders. *IEE Proceedings — Circuits, Devices and Systems*, 141(5):407–413, Oct. 1994.

[5] A. G. Dempster and M. D. Macleod. General algorithms for reduced-adder integer multiplier design. *Electronics Letters*, 31(21):1800–1802, Oct. 1995.

[6] A. G. Dempster and M. D. Macleod. Use of minimum-adder multiplier blocks in FIR digital filters. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 42(9):569–577, Sept. 1995.

[7] D. E. Goldberg. *Genetic Algorithms in search, optimization, and machine learning*. Addison-Wesley, 1989.

[8] F. Gruau. Cellular encoding as a graph grammar. In *IEE Colloquium on Grammatical Inference: Theory, Applications and Alternatives*, pages 17/1–17/10, Apr. 1993.

[9] T. Higuchi, M. Murakawa, M. Iwata, I. Kajitani, W. Liu, and M. Salami. Evolvable hardware at function level. In *IEEE International Conference on Evolutionary Computation*, pages 187–192, Apr. 1997.

[10] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley and Sons, Jan. 1979.

[11] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, 1992.

[12] J. Liang and T. D. Tran. Fast multiplierless approximations of the DCT with the lifting scheme. *IEEE Transactions on Signal Processing*, 49(12):3032–3044, Dec. 2001.

[13] M. A. Lones and A. M. Tyrrell. Enzyme genetic programming. In *Proceedings of the 2001 Congress on Evolutionary Computation*, volume 2, pages 1183–1190, May 2001.

[14] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits - part I. *Genetic Programming and Evolvable Machines*, 1(3):259–288, 2000.

[15] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802, pages 121–132, Edinburgh, 15-16 2000. Springer-Verlag.

[16] M. Schetzen. *The Volterra and Wiener Theories of Nonlinear Systems*. Krieger Publishing Company, 1980.

[17] K. Shanmugam and M. Lal. Analysis and synthesis of a class of nonlinear systems. *IEEE Transactions on Circuits and Systems*, 23(1):17–25, Jan. 1976.

[18] R. Thomson and T. Arslan. A combined evolutionary and heuristic system for the creation and optimisation of linear digital VLSI designs. *Submitted to the IEEE Transactions on Evolutionary Computation*, 2002.