SAND2000-0289 C

GECCO2000 Category: Evolvable Hardware (EH)

# Synthesis of Logic Circuits with Evolutionary Algorithms

RECEIVED FEB 17 2000 OSTI

Jake S. Jones
Evolutionary Computing Methods
Sandia National Labs, MS-0318
Albuquerque, NM 87185
jsjones@sandia.gov

George S. Davidson
Evolutionary Computing Methods
Sandia National Labs, MS-0318
Albuquerque, NM 87185
gsdavid@sandia.gov

## Abstract

In the last decade there has been interest and research in the area of designing circuits with genetic algorithms, evolutionary algorithms, and genetic programming. However, the ability to design circuits of the size and complexity required by modern engineering design problems, simply by specifying required outputs for given inputs, has as yet eluded researchers. This paper describes current research in the area of designing logic circuits using an evolutionary algorithm. The goal of the research is to improve the effectiveness of this method and make it a practical aid for design engineers. A novel method of implementing the algorithm is introduced, and results are presented for various multiprocessing systems. In addition to evolving standard arithmetic circuits, work in the area of evolving circuits that perform digital signal processing tasks is described.

## 1 INTRODUCTION

The steady increase of the power of computers has made automatic design of engineering products a reality. The ultimate goal of having a computer perform the same tasks that previously required a highly skilled engineer or computer programmer has been reached in numerous areas. As computer capacity continues climbing, more and more types of engineering problems will be solved automatically by a single engineer directing a computer tool, rather than by a team of engineers using traditional methods.

Engineering design is manpower intensive, and hence expensive. In the case of novel hardware design, automation helps by giving designers new tools that allow higher levels of expressive power in the description of their designs. For example. transistor/diode logic diagrams are less expressive than gate level drawings, which are less expressive than VLSI component level drawings, which are less expressive than VHDL design descriptions. Another boon of automation is the rule checker that catches design errors before they are built and before an error can propagate to damage other circuitry. Automation also allows synthesized circuits from VHDL. so that an abstract programming language description suffices for the design rather than schematic drawings. In all of these cases. designers must still emit designs that express their intent and many details of the designs must be specified to achieve good results.

Many applications are never developed due to the associated expense or lack of access to a designer. Ideally, many products could be built without the highest quality logic designers if the computer could synthesize simply described requirements. for example by means of frequency response functions for signal processing applications.

It is well known that analysis of the existing design is less difficult than *ab initio* synthesis of new designs, so building a design assistant. as described above. is expected to be quite difficult. Such progress as has been made has been limited to simple designs. and the technology remains too weak to address the practical design challenges faced by engineers. Design synthesis at the level of real-world problems is the focus of the work described here. The goal is to build tools to synthesize important circuits for traditional logic circuit tasks and digital signal processing applications.

In the following sections, prior efforts will be reviewed and a promising new approach to total synthesis of larger circuits will be described. Early results will be presented and analyzed. The method for producing these circuits will be described and limiting issues will be discussed. Finally, the potential for this new approach will be

## DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, make any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

## DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

explored in light of the anticipated availability of vast computing power and the need for synthesis tools that will allow modestly trained designers to produce novel, high performance products.

### 1.1 EVOLUTIONARY ALGORITHMS

One set of tools that has been shown to be effective in automatic design is the family of evolutionary algorithms. Evolutionary algorithms and genetic algorithms are two of many names that have been applied to the concept of a search algorithm with a highly random component that is patterned after Darwinian evolution. "Genetic algorithm" refers to a search space composed of a binary bitstream. "Evolutionary algorithm" refers to a search space composed of sets of integers. "Genetic Programming" means that rather than evolving the design directly, a set of instructions on how to assemble the design are evolved. With the possible exception of Genetic Programming, these terms (and numerous others not mentioned) refer to algorithms that are arguably the same idea, differing only in implementation.

These algorithms generally contain a population of competing designs. As the process progresses, the more fit designs will persist, and perhaps produce offspring that possess similar properties. The less fit designs will be replaced. The two most ubiquitous methods of generating offspring are mutation and crossover. Mutation causes a relatively small change in a design, while crossover combines two separate designs. There are countless ways of implementing these basic ideas, hence the numerous names that they have been given. An introductory description of some of these methods is found in [Mitchell,96]. Genetic Programming, another implementation of these basic ideas, is unique enough to merit separate consideration. It is described in [Koza,99].

### 1.2 LOGIC CIRCUITS

A logic circuit consists of a network of gates whose inputs and outputs are either a logic 0 or 1, and the output is a logical function of the inputs. The shape of the gate (see Figure 1) designates which logic function is implemented. The logic circuit is the basic building block of the digital computer. All digital computing, signal processing, or control reduces to a network of logic gates. The logic gates themselves are implemented using transistor logic operated in a saturated state.

Traditional design of logic circuits is by a top-down approach. An overall system design is broken down into successively smaller building blocks, until at some point the abstract blocks can be converted into a concrete logic circuit. As a result, only smaller, more manageable logic circuits must ultimately be designed. However, this comes at a price. These pre-ordained divisions, or standard design building blocks limit the optimization of the design. Subdividing a design results in a circuit that is only as good as the subdivision scheme.

### 1.3 RELATED WORK

Surveys of current work in this area can be found in [Zebulum,97] and [Yao,97]. In addition to the general survey, [Zebulum,97] describes their own work in evolving digital logic circuits. Their strategy is to evolve in software using an evolutionary algorithm, but by limiting the design space of the circuit with a preset series of gate levels: a gate may only have inputs from the previous level and may only be input to the next level.

## 2 ALGORITHM

A program was developed using C++ that uses an evolutionary algorithm to design logic circuits based on user specified input and output requirements. The input and output requirements for the circuit, that is, the method for calculating the fitness of a particular genome, must be specified by the user in a text file, which is input to the program.

### 2.1 DESCRIPTION OF GENOME

The genome of a logic circuit, i.e. the variables that must be defined to completely describe the characteristics of a single circuit, consists mainly of a variable length list of gates; each specified by five integers. The five integers specify the function of the gate, the type of each input (the output of another gate or one of the global inputs), and the index of each input. Another piece of information that is stored in the genome is a list of integers that specify which gates are connected to the global outputs. There are no preset constraints on the topology other than a maximum number of gates that can be used in a circuit.

### 2.2 TOURNAMENT SELECTION

The method of evolution consists of tournament selection followed by crossover, mutation, or randomization. The tournament consists of a number of jousts, where two contestants are selected from the population based on a weighted scale. The fitnesses of the two contestants are compared, and the loser is replaced by one of three newly generated circuits. The replacement is either a mutated version of the winner, the result of a crossover between the winner and another member of the population, or a completely random circuit.

### 2.3 PATH STRUCTURE

Each logic circuit has a skeletal path structure that aids mutation and crossover.. There are at least as many skeletal paths as there are outputs in a circuit. Each path is a route that begins at a global input, passes forward through the first input of gates and ends at an output. If there are more paths than outputs, then some of them will not end in outputs, but all outputs will be connected to a path, see Figure 1. There will be many gates that are not on a skeletal path, but are still integral to the function of the circuit.

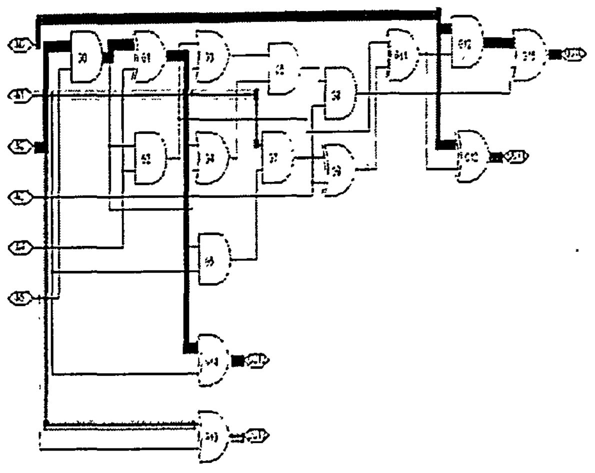


Figure 1: Evolved Circuit with 4 Outputs and 5 Paths

## 2.4 MUTATION

Circuit mutation consists of a number of individual gate mutations. An individual mutation is one of 5 things: adding a gate, changing a gate function, changing a gate's first input, changing a gate's second input, or removing (shorting) a gate. The number of individual mutations that occur in a single mutation step is a random number from one to some maximum number of mutations. The maximum number of mutations is a variable that is increased during lulls in the progress of the evolutionary algorithm, to help escape fitness plateaus.

The gate addition mutation places a random type gate at the end of one of the paths. The first input is connected to the output of the end of the path, and the second input (if it exists) is connected to another randomly selected gate in the circuit. The new gate then becomes the end of the path.

Shorting a gate removes a randomly selected gate, splicing the output to the first input, and dropping the second input connection. Changing a gate's input refers to randomly choosing another input gate (or global input) without regard to the skeletal path structure. Changing a gate function simply picks randomly from the list of possible types of gates.

## 2.5 CROSSOVER

With crossover, the genome of the winner is mixed with another randomly selected genome (weighted scale), and this new genome replaces the loser of the tournament. The method of mixing the circuits guarantees that the individual fitness performance of each output of the new circuit is exactly like the fitness performance for the same output of one of the two parents. Which parent each part of the circuit comes from is a random variable. For example, crossover can create a circuit whose performance for outputs 1, 2, and 4 is identical to the first parent, and whose performance for the remaining outputs is the same as the second parent. In this manner, the crossover algorithm creates circuits that are related to the parents in both fitness performance and internal structure, and avoids the unpredictable fitness performance that results from splicing two circuits together based on a random slice through each circuit.

## 2.6 MODULARITY

The object-oriented design of the system makes changing the type of circuit evolved very easy. The possible types of logic gates (AND, OR, XOR etc.) can be changed according to the available gate functions on the hardware for which the circuit is destined. Different types of logic circuits can be evolved by setting the parameter that selects the fitness evaluation function. Furthermore, the system is not limited to logic circuits. Any circuit structure could be designed by creating an appropriate module, for instance, analog circuits or neural networks.

## 2.7 OTHER FEATURES

Of the many other features in the system, one of the most useful is the ability to periodically write the best genomes out to files, and then to use these genomes as seeds to start new simulations. Other capabilities include optimizing the size of the circuit and writing out history data.

## 2.8 COMPUTE PLATFORMS

The single processor version runs with a menu system on WindowsNT or from a command line on any Unix platform. The parallel version has been run on a 232 processor Linux cluster and a 143 processor WinNT cluster supercomputer. The parallelization (using MPI) of the code is straightforward. Each node has its own population and its own simulation, but at the end of each timestep a random segment of each population emigrates to the next processor node in the loop. In this fashion evolutionary progress is communicated between the nodes.

## 3 APPLICATIONS

### 3.1 ARITHMETIC CIRCUITS

One category of circuit that has been evolved with the system is a simple feed-forward arithmetic logic circuit.

In this case, the term feed-forward is used to describe a circuit that has no feedback loops and no memory. When a constant input is applied to the system, the output will remain constant after a certain number of clock cycles. This rule is enforced by the requirement that no gate can be an input to a gate with a lower index number. The input files specify the fitness function by listing the expected output per input. Inappropriate inputs (or don't-cares) are not included in the input file. Cases where some of the outputs are don't-cares can also be handled.

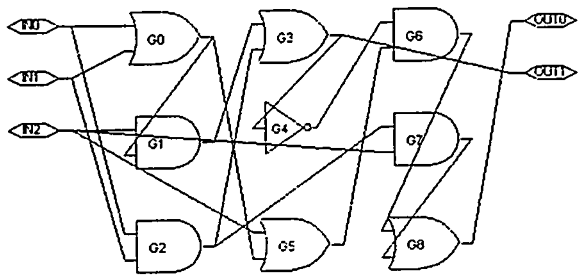


Figure 2: Example of an Evolved Adder (1bit+1bit+carry)

Figure 2 shows a simple, evolved adder (1bit+1bit+carry bit) that was created in less than a minute on a single processor, using only one or two input AND, OR and INV gates. Figure 3 shows the equivalent standard sum-of-products realization of the same circuit, using the same types of gates. The sum-of-products realization is not a guaranteed minimal implementation, but it is a textbook approach to creating logic circuits that implement Boolean truth tables, so it is a meaningful comparison.

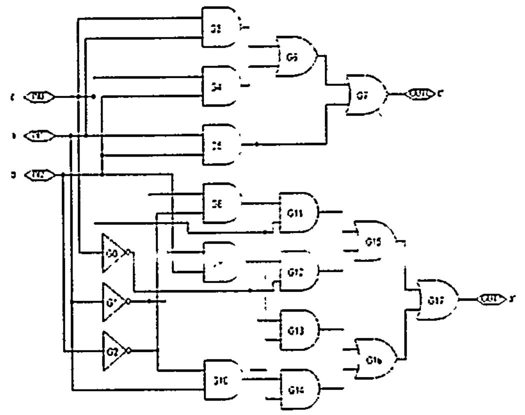


Figure 3: Textbook Design Sum-of-Products Adder (1bit+1bit+carry)

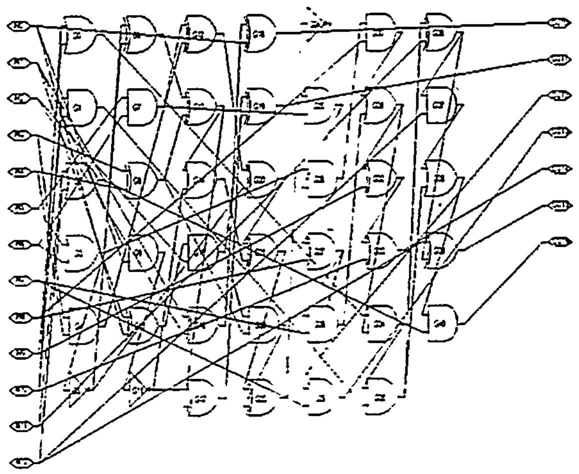
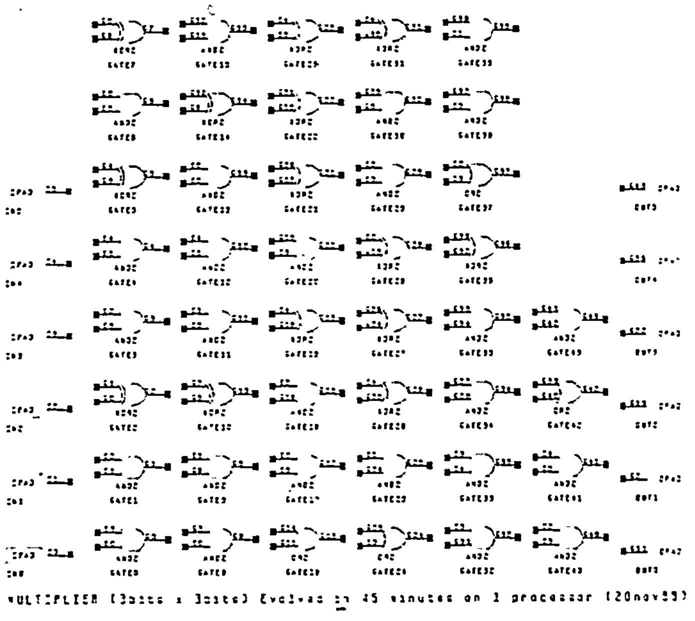


Figure 4: Larger Example of an Evolved Adder (6bit+6bit−carry)

Figures 4 and 5 show an evolved adder (6bits+6bits+carry bit) and multiplier (3bits*3bits). These two circuits were evolved with the following set of possible gates: AND, OR, XOR, and INV. Table 1 shows part of the input file for the 6 bit adder problem. These two circuits are some of the largest arithmetic circuits evolved with this system to date. The adder took 1 hour to evolve, using 128 processors. The multiplier evolved in 45 minutes on a single processor.


MULTIPLIER (3bits x 3bits) Evolved in 45 minutes on 1 processor (20nov95)
Figure 5: Example of an Evolved Multiplier (3bit*3bit)

```
!!!!
!!!! input order:
!!!! c a0 b0 a1 b1 a2 b2 a3 b3 a4 b4 a5 b5
!!!! output order: a0 a1 a2 a3 a4 a5 c
!!!!
!!! max_num_mutations max_num_gates population time_limit
16 60 400 20000
!!! num_ins  num_outs num_fitness_sets
13 7 1
!!!
!!!weights
1
1
1
1
1
1
1
!!!num_test_cases for fitness set 1
8192
!!!!n plus n plus carry: inputs
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0
......
......
!!!!n plus n plus carry: outputs
0 0 0 0 0 0 0
1 0 0 0 0 0 0
0 1 0 0 0 0 0
1 1 0 0 0 0 0
0 0 1 0 0 0 0
......
......
```

Table 1: Part of an Input File

## 3.2 DISCRETE-TIME DIGITAL FILTERS

### 3.2.1 Finding Filter Weights Using Evolutionary Algorithms

Before discussing the evolution of logic circuits that perform discrete-time digital filtering, the simpler task of finding filter weights should be briefly mentioned. This is a problem that has been discussed in the literature in numerous guises. Given a fixed topology of some calculation graph, the problem is to use a search algorithm to find an optimal set of weights for the nodes of the graph. This approach has been used for everything from finding a polynomial that fits arbitrary data to finding the weights of a fixed neural network. The subject of this section, however, is the case where the topology is *not only unfixed, but also unlimited.*

### 3.2.2 Digital Signal Processing

The second category of circuit that has been evolved with the system is a discrete-time digital filter, in which the logic circuit must retain a memory of past inputs. In this type of problem, the input is a series of values representing discrete moments in time of a time-varying signal. The output is another series of values that are a filtered result of the input values. A Low-Pass Filter (LPF) stops all high frequency components of a signal and lets lower frequency components pass through. The second graph of Figure 6 shows the frequency domain transfer function of a 16th order Hamming window low pass filter, including positive and negative frequencies, with the center representing zero frequency.

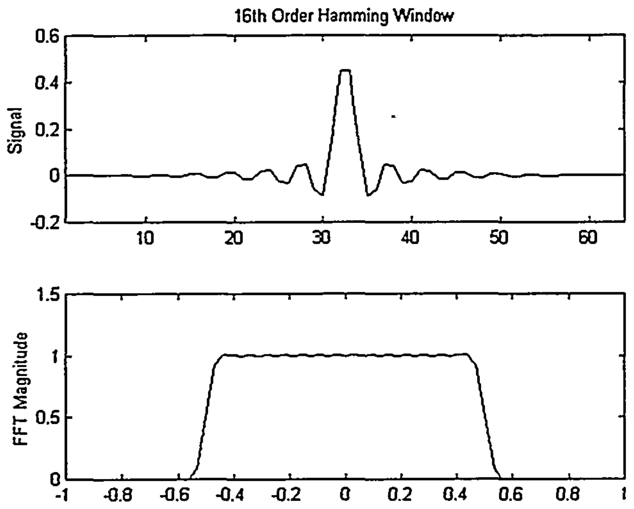


FIGURE 6: A 16th order Hamming window LPF

In order to evolve a logic gate network that can play the part of an LPF, a method for evaluating the fitness of the circuit is needed. The most obvious method would be to input white noise into the circuit, take the Fast Fourier Transform (FFT) of the output, and compare it's magnitude to the desired transfer function. Better circuits would then have more of the shape of an LPF, as in the second graph of Figure 6. The fitness could be the sum of the errors between the desired transfer function and the measured one, taken at discrete points in the frequency domain. To strengthen the performance of the circuit in the transition band (the transition area between no-pass and all-pass) or any other area, more measurements could be taken in that zone.

The problem with this approach is the cost of calculating an FFT. The number of calculations required for an FFT is proportional to *NlogN*, where *N* is the number of data points. This problem is compounded by the fact that, due to the nonlinearity of a network of logic gates, many data points will be required in order to assure that the circuit performs as expected with all inputs.

Another method is to calculate the fitness without using the FFT. A training set of white noise inputs with their corresponding filtered outputs (calculated in advance) would be used to evaluate circuits in the population. Evaluating the circuits in this manner would be much faster than the FFT, but there are some clear disadvantages. The first is that pre-generated noise sequences would be used, which could lead to circuits that are tuned for the characteristics of those sequences. Using the FFT method, however, new noise could be generated at every turn. The second disadvantage would be in the calculation of the actual fitness value. Summing the differences in the time domain will not be as effective for this type of problem, where the main characteristics of interest are in the frequency spectrum.

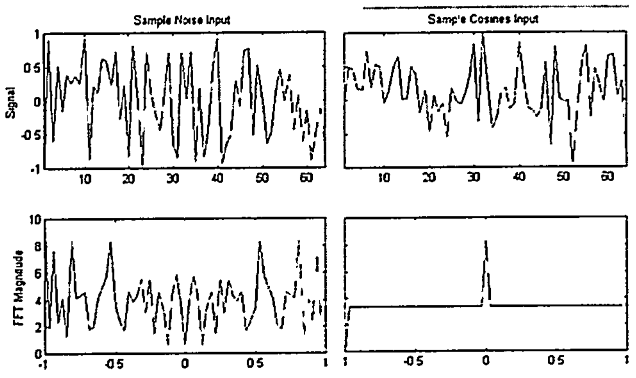


FIGURE 7: Sample Training Data

Having selected the second method, that is, evaluating the circuit only in the time domain, the next question is how to implement the process? Two types of input data were used: white noise and a set of random phase cosines that cover the spectrum. For the latter input, a random-phase cosine is added at each integral multiple of the fundamental frequency. The top graphs of Figure 7 show samples of the two types of input and the bottom graphs show their frequency spectra. These input streams, along with their desired outputs, are then converted into two's complement 8-bit integers for use with an 8-bit input and 8-bit output network of logic gates. The weights of the bits are specified in the program input file: the least significant bit has a weight of one, and then the next bit has a weight of two, doubling each time until the sign bit, which has a weight of 128. In this fashion, the fitness function provides more incentive to get the higher order bits correct, which should lead to an acceptable answer much faster.

### 3.2.3 Register Bits

A new gate type was added to allow the system to evolve circuits with memory: a register bit. In the software implementation of this type of circuit, all register bits are evaluated as if cued by a clock pulse, so that the output of the register bit will change only once per evaluation. In this manner signals will not necessarily propagate all the way to the output bits during a single evaluation of the output.

### 3.2.4 Results

To test this method, a circuit that implements a 16th order Hamming window LPF for an 8-bit input was evolved. The simulation was run using 128 processors of a WinNT cluster. The training data consisted of twenty white noise sequences of 64 samples each, along with their desired solutions. At various times in the simulation, the best evolved circuit was copied and then examined in a separate program, by testing it's performance upon another 200 newly generated white noise inputs.

The graph of Figure 8 shows three different lines. The top line is the theoretical average output error for a randomly generated circuit. The middle line is the average output error for a circuit that was evolved in one hour, consists of 1307 gates, and has a fitness of 25.40% (where 0% is a perfect circuit). The bottom line is the error of a circuit that was evolved in four hours, consists of 1933 gates, and has a fitness of 17.84%. This figure shows clearly that the process is working, and that as evolution progresses, the output of the circuits is closer to the desired output.

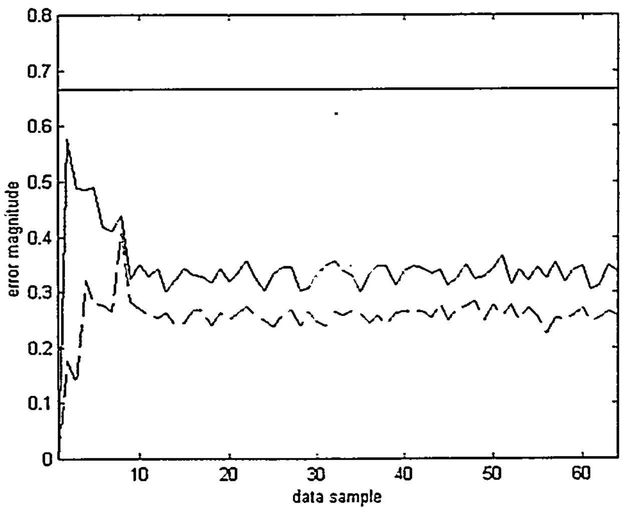


FIGURE 8: Error of Evolved LPF Circuits

Figure 9 shows the magnitude of the frequency spectrum for the evolved circuits. The top solid line shows the average spectrum of the 200 sets of input data. As expected from white noise, it is mostly flat. The bottom solid line shows the desired spectrum, i.e. the average spectrum of the results of applying a true 16th order Hamming window LPF to each of the sets of input data. The remaining lines are the spectra of the evolved circuits. The earlier circuit is already showing some lowering in the higher frequencies. The later circuit shows further attenuation in the higher frequencies, and is beginning to take the shape of an LPF.

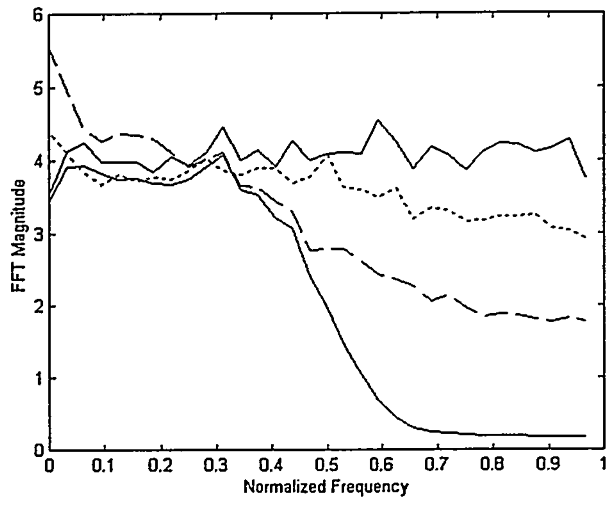


FIGURE 9: FFT Magnitude of Evolved LPF Circuits

These are very promising results: a logic circuit was evolved that performs a digital signal processing task that is well within the realm of what was earlier called a "real-world problem". However, there are points that need further investigation before this system can be a practical signal processing design tool. The most pressing task remaining is to investigate how many different noise data points are required in the fitness data in order to guarantee that the evolved circuit will be able to handle arbitrary data. If only a single 64-sample dataset is used, it was found that the evolved circuit performs the task well only on that dataset. Using twenty datasets led to the successful results shown earlier, but the question remains as to whether further evolution would improve the circuit or cause it to become particularly tuned to the twenty datasets used.

### 3.2.5 Related Work

Other research in this area is described in [Miller,99]. The approach used was to calculate the FFT of the output data and evaluate the circuits in the frequency domain. The inputs were single zero-phase sine waves at integral multiples of the fundamental frequency of the data. However, the circuits were not subjected to noise inputs, or non-zero-phase sines. The resulting circuits were successful in processing individual sines, but did not handle linear combinations of sines well.

## 4 EVOLVABLE HARDWARE

### 4.1 EVALUATING CIRCUITS IN HARDWARE

Another goal of this work was to use a Field Programmable Gate Array (FPGA) to evaluate circuits in the population by programming them onto the chip and evaluating them directly. The idea of using re-programmable logic and evolutionary algorithms to design static or self-correcting logic circuits has been known as Evolvable Hardware. An example of this can be found in [Thompson, 97], where evolution occurred at the lowest level, that is, where FPGA cell functions, connectivity, and routing were all subject to evolutionary change. For the work described in this paper, however, it was desired that the routing not be a factor in the evolution. If the routing is not a factor then the circuits evolved will only contain the abstract connectivity information of the circuit. In this fashion, it is possible to evolve conjointly in software and in hardware without requiring the software to perform routing functions.

A method of quickly programming abstract logic circuits into an FPGA, without having to route them, was required. A general logic circuit composed of multiplexers as inputs to basic logic gates was designed and routed. Associated with each gate in the abstract netlist are three registers. One register determines the function of the gate (AND, OR, XOR, etc), while the other two determine from which gates the inputs are read. A feedforward-only multiplexor circuit with a maximum of 46 gates was designed, routed and loaded onto a Xilinx XC4000 FPGA [Xilinx, Inc.] mounted on a FAT-HOTWorks PCI-Board [Virtual Computer Corporation, Inc] in a PC running Windows95.

With this multiplexor circuit loaded into the FPGA, any abstract logic circuit (up to a size limit) can be implemented with the FPGA simply by writing to three registers per gate, with no routing required. The largest abstract circuit that can be programmed in this fashion has 46 gates, 8 inputs and 8 outputs. Once the multiplexor circuit is programmed to implement a particular abstract design, then for each input that needs to be tested, the data is written to the 8-bit input register and then read from the 8-bit output register.

The motivation for evaluating a circuit directly in the hardware rather than in a simulation of a logic circuit is speed. A software simulation of a feedforward logic circuit must loop forward once through all of the gates, calculating each output based on it's logic function and the outputs of the previous gates. The time required for this loop is assumed to be proportional to the number of gates. With the hardware implementation there is an initial up-front time required to program the circuit. Once this is done, the remaining time required is independent of the number of gates. This is true because a signal propagates through the multiplexor network much faster than the time required to execute the several assembly language commands that access the input and output registers through the HOTWorks PCI Board. In other words, the time required to evaluate any circuit is equal to the time needed to write the input register plus the time needed to read the output register.

There is a number of gates that is the break-even point, where the software evaluation time is the same as the hardware evaluation time. In this case, on the 95Mhz Pentium with the HOTWorks card installed, the approximate break-even point was twelve gates (with 256 input/output fitness data pairs). For a circuit of maximum size (46 gates), with 256 input/output fitness data pairs, there was approximately a 4x speedup by evaluating the circuits in hardware rather than software.

## 5 CONCLUSIONS

This paper described current work in the area of synthesizing logic circuits using evolutionary algorithms, in both software and hardware. The problem is of interest due to the potential to reduce the required number of trained engineers for project design. An algorithm was shown, as well as examples of its use, taking advantage of both parallel processing and hardware acceleration with an FPGA. Use of the FPGA was facilitated by the design of a circuit that can implement any abstract circuit by simply writing to registers that specify connectivity.

In the area of arithmetic circuits, results were shown that are as large or larger than others seen in similar research, though arguably still not large enough to be an effective design aid. In the novel area of synthesizing logic circuits to perform digital signal processing tasks, results were shown to surpass the literature. Circuits were evolved that perform rough low pass filtering according to user specifications. In this area the process shows great potential.

**References**

J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving,*Morgan Kaufmann Publishers, 1999.

J. F. Miller, "Digital Filter Design at Gate-level using Evolutionary Algorithms", Proceedings of the Genetic and Evolutionary Computation Conference, Vol. 2, pp.1127-1134, Morgan Kaufmann, 1999.

M. Mitchell, *An Introduction to Genetic Algorithms,* MIT Press, 1996.

A. Thompson, "An Evolved Circuit, Intrinsic in Silicon, Entwined with Physics", Lecture Notes in Computer Science - Evolvable Systems: From Biology to Hardware, Vol. 1259, pp. 390-405, Springer-Verlag, 1997.

Virtual Computer Corportation, Inc. See http://www.vcc.com

Xilinx, Inc. See http://www.xilinx.com

X. Yao, T. Higuchi, "Promises and Challenges of Evolvable Hardware", Lecture Notes in Computer Science - Evolvable Systems: From Biology to Hardware, Vol. 1259, pp. 55-78, Springer-Verlag, 1997.

R. S. Zebulum, M. A. Pacheco, M. Vellasco, "Evolvable Systems in Hardware Design: Taxonomy, Survey and Applications", Lecture Notes in Computer Science - Evolvable Systems: From Biology to Hardware, Vol. 1259, pp. 344-358, Springer-Verlag, 1997.

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.