Evolving Multiplier Circuits by Training Set and Training Vector Partitioning

Jim Torresen

Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
jimtoer@ifi.uio.no
http://www.ifi.uio.no/~jimtoer

Abstract. Evolvable Hardware (EHW) has been proposed as a new method for evolving circuits automatically. One of the problems appearing is that only circuits of limited size are evolvable. In this paper it is shown that by applying an approach where the training set as well as each training vector is partitioned, large combinational circuits can be evolved. By applying the proposed scheme, it is shown that it is possible to evolve multiplier circuits larger then those evolved earlier.

1 Introduction

For the recent years, evolvable hardware (EHW) has become an important scheme for automatic circuit design. However, still there lack schemes to overcome the limitation in the chromosome string length [1, 2]. A long string is required for representing a complex system. However, a larger number of generations are required by genetic algorithms (GA) as the string length increases. This often makes the search space *too* large and explains why only small circuits have been evolvable so far. Thus, work has been undertaken trying to diminish this limitation. Various experiments on *speeding up* the GA computation have been undertaken [3]. The schemes involve fitness computation in parallel or a partitioned population evolved in parallel. Experiments are focussed on speeding up the GA computation, rather than dividing the application into subtasks. This approach assumes that GA finds a solution if it is allowed to compute enough generations. When small applications require weeks of evolution time, there would probably be strict limitations on the systems evolvable even by parallel GA.

Other approaches to the problem have used variable length chromosomes [4]. Another option, called function level evolution, is to apply building blocks more complex than digital gates [5]. Most work is based on fixed functions. However, there has been work in Genetic Programming for *evolving* the functions [6]. The method is called Automatically Defined Functions (ADF) and is used in software evolution.

An improvement of artificial evolution — called co-evolution, has been proposed [7]. In co-evolution, a part of the data which defines the problem co-evolves simultaneously with a population of individuals solving the problem. This could lead to a solution with a better generalization than a solution based only on the initial data. A variant of co-evolution — called cooperative co-evolutionary algorithms, has been proposed by De Jong and Potter [8, 9]. It consists of parallel evolution of sub-structures which interact to perform more complex higher level structures. Complete solutions are obtained by assembling representatives from each group of sub-structures together. In that way, the fitness measure can be computed for the top level system. However, by testing a number of individuals from each sub-structure population, the fitness of individuals in a sub-population can be sorted according to their performance in the top-level system. Thus, no explicit local fitness measure for the sub-populations are applied in this approach. However, a mechanism is provided for initially seeding each GA population with user-supplied rules. Darwen and Yao have proposed a co-evolution scheme where the subpopulations are divided without human intervention [10].

Incremental evolution for EHW was first introduced in [11] for a character recognition system. The approach is a divide-and-conquer on the evolution of the EHW system, and thus, named *increased complexity evolution*. It consists of a division of the *problem* domain together with incremental evolution of the hardware system. Evolution is first undertaken individually on a set of basic units. The evolved units are the building blocks used in further evolution of a larger and more complex system. The benefits of applying this scheme is both a *simpler* and *smaller* search space compared to conducting evolution in one single run. The goal is to develop a scheme that could evolve systems for complex real-world applications. There has been undertaken work on decomposition of logic functions by using evolution [12]. Results on evolving a 7-input and 10-output logic function show that such an approach is beneficial.

The largest correctly working multiplier circuit evolved – the author is aware of, is evolved by Vassilev et al [13]. It is a 4×4-bit multiplier evolved in a single run (8 outputs) by gates as building blocks. In this paper, the multiplier will not be evolved in a single run but rather be evolved as separate subsystems that together perform a correct multiplication. Normally, evolution is used to evolve a circuit with a smallest possible number of gates. In this work, we rather concentrate on reducing the evolution time at the same time as being able to solve problems not evolvable in a single run. This is motivated by the fact that the number of transistors becoming available in circuits increases according to Moores Law. Thus, this approach is based on "wasting" transistors to faster evolve a circuit or evolve circuits than are *not* evolvable in a single run.

The experiments are undertaken for evolving a 5×5-bit multiplier circuit. There is a substantial increase in complexity between a four-by-four bit multiplier (256 lines in the truth table) compared to a five-by-five bit multiplier (1024 lines in the truth table). Even though the experiments are for evolving multiplier circuits, the principles are valid for any combinational circuit defined by a complete truth table.

The next section introduces the concepts of the multiplier evolution. Results of experiments are reported in Section 3 with conclusions in Section 4.

2 Multiplier Evolution by Data Partitioning

In this section, the *increased complexity evolution* is applied to evolve a large multiplier circuit.

2.1 Approaches to Increased Complexity Evolution

Several alternative schemes on how to apply this method are present:

- **Partitioned training vector.** A first approach to incremental evolution is by partitioning each training vector. For evolving a truth table - i.e. like those used in digital design, each output could be evolved separately. In this method, the fitness function is given explicitly as a subset of the complete fitness function.
- **Partitioned training set.** A second approach is to divide the training set into several subsets. For evolving a truth table, this would correspond to distributing the rows into subsets, which are evolved separately [14]. This would be similar to the way humans learns: Learning to walk and learning to talk are two different learning tasks. The fitness function would have to be designed for each task individually and used together with a global fitness function. Later in this paper it is shown how this can be done in digital design.

In this paper, these two approaches are combined.

X * **Y** = **Z**
00 * 00 = 0000
00 * 01 = 0000
00 * 10 = 0000
00 * 11 = 0000
01 * 00 = 0000
01 * 01 = 0001
01 * 10 = 0010
01 * 11 = 0011
10 * 00 = 0000
10 * 01 = 0010
10 * 10 = 0100
10 * 11 = 0110
11 * 00 = 0000
11 * 01 = 0011
11 * 10 = 0110
11 * 11 = 1001

Table 1. The truth table for multiplying two by two bits.

Table 1 shows the truth table for a two by two bit multiplier circuit. In the context of evolving multiplier circuits, the training set consists of all possible permutations of the inputs. Evolving this circuit in a single run requires a circuit with four inputs and four outputs. If we instead would like to apply the *increased complexity evolution* we could have a partitioning like the one given in Table 2. This is for illustration only since there would be no problem in evolving such a small truth table in a single run. In this case, we would evolve a separate circuit for each of the four outputs – partitioned training vector (PTV). Further, partitioned training set (PTS) is also applied and divides the training set into two parts. This means that we end up with evolving eight subsystems in total (PTV equal to four multiplied by PTS equal to two). The most significant input bit (x_1) is used to distinguish between the two training set partitions. Thus, the final system can be assembled together – as seen in Fig. 1, to perform a correct multiplier function. This requires during evolution that the truth table vectors are assigned into two groups each applied to a evolve its separate groups of subsystems (A and B) as the horisontal line (and x_1) in Table 2 indicates. Within each such group there is no demand for sorting the vectors.

x_1x_0	*	y_1y_0	z_3	z_2	z_1	z_0
00	*	00	0	0	0	0
00	*	01	0	0	0	0
00	*	10	0	0	0	0
00	*	11	0	0	0	0
01	*	00	0	0	0	0
01	*	01	0	0	0	1
01	*	10	0	0	1	0
01	*	11	0	0	1	1
10	*	00	0	0	0	0
10	*	01	0	0	1	0
10	*	10	0	1	0	0
10	*	11	0	1	1	0
11	*	00	0	0	0	0
11	*	01	0	0	1	1
11	*	10	0	1	1	0
11	*	11	1	0	0	1

Table 2. The truth table for multiplying two by two bits.

The amount of hardware needed would be larger than that needed for evolving one system in a single run. However, the important issue – as will be shown in this paper, is the possibility of evolving circuits *larger* than those evolved by a single run evolution.

The number of generations required to evolve a given subsystem would be highly dependent on the training vectors for the given training set partition. Thus, to minimize both the evolution time and the amount of logic gates, an algorithm should be applied to find both the partitioning size of the training set as well as the partitioning of each training vector. The best way this can be undertaken is probably by starting with a fine grain partitioning to be able to find a working system – even though the amount of logic would be large. Then, the training vectors for consecutive subsystems that was easily evolved

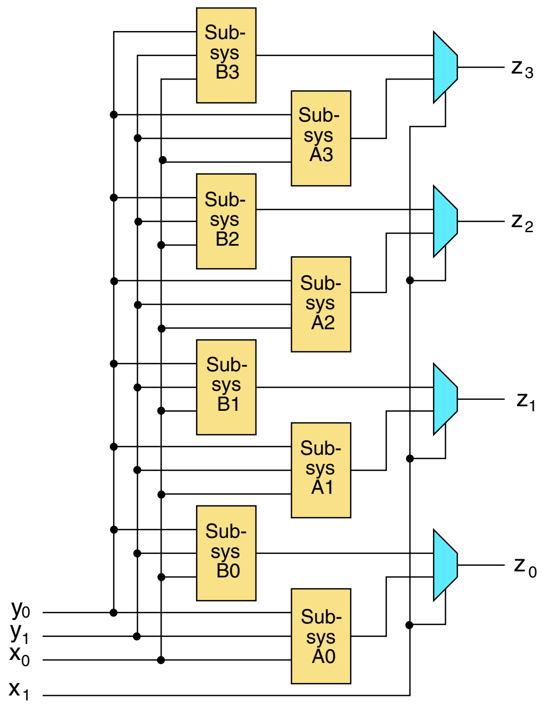

Fig. 1. An assembly of subsystems.

can be *merged* and applied to evolve *one* new subsystem that can substitute the group of subsystems earlier evolved. This, will reduce the hardware size. Another optimizing feature to be included is to look into the training set from the beginning to make dedicated partitions. That is, if a set of consecutive input vectors result in the same output value, there is no need to actually evolve a subsystem for this. In the following experiments we apply fixed partitioning sizes, however results will probably indicate that it will be helpful with an adaptive partitioning.

2.2 Target Hardware

Each subsystem evolved consists of a fixed-size array of logic gates. The array consists of n gate layers from input to output as seen in Fig. 2. Each logic gate (LG) is either an *AND* or an *XOR* gate. Each gate's two inputs in layer l is connected to the outputs of two gates in layer $l-1$. In the following experiments, the array consists of 8 layers each consisting of 16 gates. A 5×5-bit multiplier consists of 10 outputs which we are going to evolve by partitioning as described in the example above. Thus, only one gate is used in the layer 8 of the gate array. This results in a maximum number of gates for each subsystem: $N_{ss} = 4 \cdot 16 + 8 + 4 + 2 + 1 = 79$ gates.

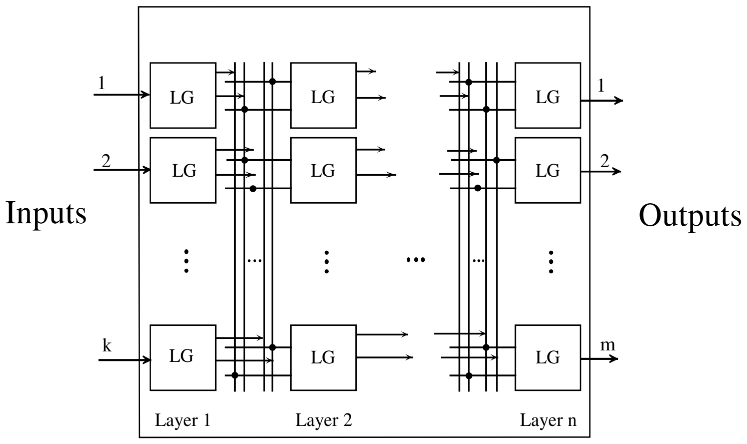

Fig. 2. The architecture of a gate array subsystem.

The *function* of each gate and its *two inputs* are determined by evolution. The encoding of each logic gate in the chromosome string is as follows:

Input 1 (4 bit)	Input 2 (4 bit)	Function (1 bit)

For the given array the chromosome string length becomes 1017 bit long which earlier experiments have shown to be appropriate.

2.3 GA Parameters and Fitness Function

Various experiments were undertaken to find appropriate GA parameters. The ones that seemed to give the best results were selected and fixed for all the experiments. This was necessary due to the large number of experiments that would have been required if GA parameters should be able vary through all the experiments. The preliminary experiments indicated that the parameter setting was not a major critical issue.

The simple GA style – given by Goldberg [15], was applied for the evolution with a population size of 50. For each new generation an entirely new population of individuals is generated. Elitism is used, thus, the best individuals from each generation are carried over to the next generation. The (single point) crossover rate is 0.5, thus the cloning rate is 0.5. Roulette wheel selection scheme is applied. The mutation rate – the probability of bit inversion for each bit in the binary chromosome string, is 0.005.

The fitness function is computed in the following way:

$$F = \sum_{\text{vec}} \sum_{\text{outp}} x \qquad \text{where } x = \begin{cases} 0 \text{ if } y \neq d \\ 1 \text{ if } y = d = 0 \\ 2 \text{ if } y = d = 1 \end{cases} \tag{1}$$

For each output the computed output y is compared to the target d. If these equal and the value is equal to zero then 1 is added to the fitness function. On the other hand, if they equal and the value is equal to one, 2 is added. In this way, an emphasize is given to the outputs being one. This has shown to be important for getting faster evolution of well performing circuits. The function sum these values for the assigned outputs (outp) for the assigned truth table vectors (vec).

The proposed architecture fits into most FPGAs (Field Programmable Gate Arrays). The evolution is undertaken off-line using software simulation. However, since no feed-back connections are used and the number of gates between the input and output is limited, the real performance should equal the simulation. Any spikes could be removed using registers in the circuit.

Due to a large number of experiment (some requiring many generations to evolve) there has so far only been time to evolve one run of each reported experiment.

3 Results

This section reports the experiments undertaken to evolve multiplier circuits in an efficient way.

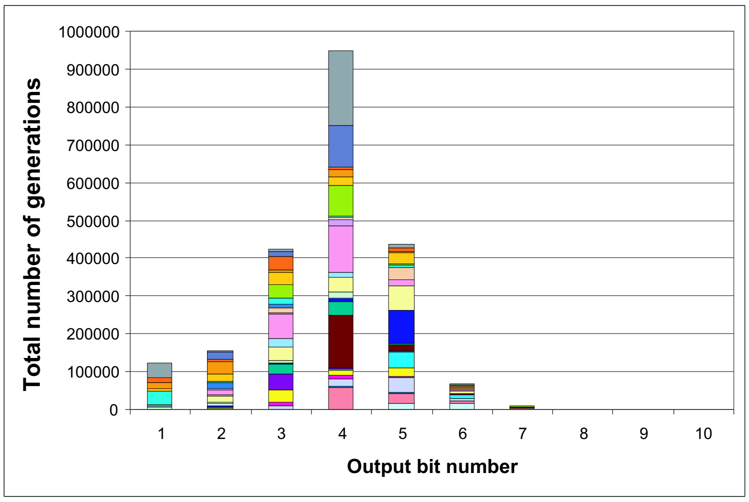

Fig. 3. Results of evolving one output at a time with 32 training set partitions (each column sums the number of generations required to evolve each output bit).

The experiments are based on using fixed partitioning sizes of the training set as well as evolving for one output bit at a time. Fig. 3 shows the results of evolving with 32 training set partitions. That correspond to 32 vectors from the truth table being used for evolving each subsystem. Each stacked column is the sum of the number of generations used for evolving subsystems for each of the 32 partitions. Output bit number 1 is the most significant bit while bit number 10 is the least significant bit. For every partition it was possible to find a correctly working subsystem. This was by a single[1] run of each subsystem evolution, rather than a selection among multiple runs. Thus, the experiment has shown that by the given scheme it is possible to evolve multipliers larger that those reported earlier. There should be no limit in applying this approach for evolving even larger multiplier circuits. However, the amount of hardware will also probably then have to be substantially increased.

It is most difficult to evolve the five most significant bits with bit four requiring the largest number of generations (close to 1,000,000 generations). Bit 10 is the easiest to evolve and requires only 102 generations in total. This latter case would be a waste of gate resources since it (in another experiment) was evolved in a single run in 71 generations That is, this system was evolved without training set partitions at all. The results indicate the need for an adaptive size in training set partitioning.

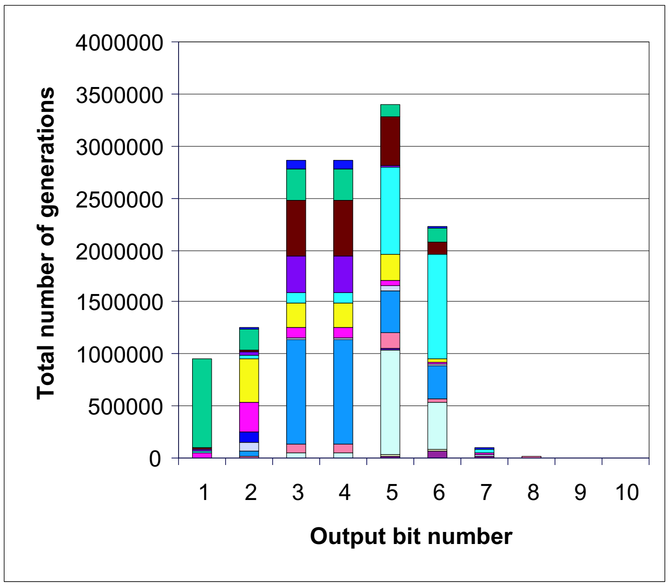

Fig. 4. Results of evolving one output at a time with 16 training set partitions (each column sums the number of generations required to evolve each output bit).

[1] With one exception – that reached the maximum 200,000 generations. It was easily evolved in a second run.

To show how the “difficult” parts of the truth table become harder to evolve with a smaller number of traning set partitions, an experiment with 16 training set partitions was conducted. As seen in Fig. 4, output bit 3, 4 and 5 now require close to 3 million generations to evolve. Moreover, there was four instances (one for each output bit 3, 4, 5 and 6) where no correctly working circuit was found in 1 million generations. Thus, here we see the need for small partioning sizes to be able to find a correctly working circuit.

Even though it has been shown successful to evolve a large multiplier circuit, a 5×5 multiplier circuit is still much smaller than what can be designed in the traditional way. Thus, future work should concentrate on researching extension of this and other techniques for achieving automatic design of even larger logic circuits.

4 Conclusions

This paper has presented how incremental evolution can be applied for evolving multiplier circuits. The scheme is focused on evolution time and evolvability rather than minimizing hardware. Experiments verify that this is beneficial for solving more complex problems. A 5×5 multiplier circuit was evolved which is larger than any other reported evolved multiplier circuit.

References

1. W-P. Lee, J. Hallam, and H.H. Lund. Learning complex robot behaviours by evolutionary computing with task decomposition. In A. Birk and J. Demiris, editors, *Learning Robots: Proc. of 6th European Workshop, EWLR-6 Brighton*, volume 1545 of *Lecture Notes in Artificial Intelligence*, pages 155–172. Springer-Verlag, 1997.
2. X. Yao and T. Higuchi. Promises and challenges of evolvable hardware. In T. Higuchi et al., editors, *Evolvable Systems: From Biology to Hardware. First International Conference, ICES 96*, volume 1259 of *Lecture Notes in Computer Science*, pages 55–78. Springer-Verlag, 1997.
3. E. Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.
4. M. Iwata, I. Kajitani, H. Yamada, H. Iba, and T. Higuchi. A pattern recognition system using evolvable hardware. In *Proc. of Parallel Problem Solving from Nature IV (PPSN IV)*, volume 1141 of *Lecture Notes in Computer Science*, pages 761–770. Springer-Verlag, September 1996.
5. M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at function level. In *Proc. of Parallel Problem Solving from Nature IV (PPSN IV)*, volume 1141 of *Lecture Notes in Computer Science*, pages 62–71. Springer-Verlag, September 1996.
6. J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* The MIT Press, 1994.
7. W.D. Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1-3):228–234, 1990.

8. K.A. De Jong and M.A. Potter. Evolving complex structures via co-operative coevolution. In *Proc. of Fourth Annual Conference on Evolutionary Programming*, pages 307–317. MIT Press, 1995.
9. M.A. Potter and K.A. De Jong. Evolving neural networks with collaborative species. In *Proc. of Summer Computer Simulation Conference.* The Society for Computer Simulation, 1995.
10. P. Darwen and X. Yao. Automatic modularization by speciation. In *Proc. of 1996 IEEE International Conference on Evolutionary Computation*, pages 88–93, 1996.
11. J. Torresen. A divide-and-conquer approach to evolvable hardware. In M. Sipper et al., editors, *Evolvable Systems: From Biology to Hardware. Second International Conference, ICES 98*, volume 1478 of *Lecture Notes in Computer Science*, pages 57–65. Springer-Verlag, 1998.
12. T. Kalganova. Bidirectional incremental evolution in extrinsic evolvable hardware. In J. Lohn et al., editor, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 65–74. IEEE Computer Society, Silicon Valley, USA, July 2000.
13. D. Job V. Vassilev and J. Miller. Towards the automatic design of more efficient digital circuits. In J. Lohn et al., editor, *Proc. of the 2nd NASA/DoD Workshop on Evolvable Hardware*, pages 151–160. IEEE Computer Society, Silicon Valley, USA, July 2000.
14. J. F. Miller and P. Thomson. Aspects of digital evolution: Geometry and learning. In M. Sipper et al., editors, *Evolvable Systems: From Biology to Hardware. Second International Conference, ICES 98*, volume 1478 of *Lecture Notes in Computer Science*, pages 25–35. Springer-Verlag, 1998.
15. D. Goldberg. *Genetic Algorithms in search, optimization, and machine learning.* Addison–Wesley, 1989.