Scalability Problems of Digital Circuit Evolution
Evolvability and Efficient Designs

Vesselin K. Vassilev
School of Computing
Napier University
Edinburgh, EH14 1DJ, UK
Tel: ++44(0)131-455 4432
E-mail: v.vassilev@dcs.napier.ac.uk

Julian F. Miller
School of Computer Science
University of Birmingham,
Birmingham, B15 2TT, UK
Tel: ++44(0)121-414 3710
E-mail: j.miller@cs.bham.ac.uk

Abstract

A major problem in the evolutionary design of combinational circuits is the problem of scale. This refers to the design of electronic circuits in which the number of gates required to implement the optimal circuit is too high to search the space of all designs in reasonable time, even by evolution. The reason is twofold: firstly, the size of the search space becomes enormous as the number of gates required to implement the circuit is increased, and secondly, the time required to calculate the fitness of a circuit grows as the size of the truth table of the circuit.

This paper studies the evolutionary design of combinational circuits, particularly the three-bit multiplier circuit, in which the basic building blocks are small sub-circuits, modules inferred from other evolved designs. The structure of the resulting fitness landscapes is studied and it is shown that in general the principles of evolving digital circuits are scalable. Thus to evolve digital circuits using modules is faster, since the building blocks of the circuit are sub-circuits rather than two-input gates. This can also be a disadvantage, since the number of gates of the evolved designs grows as the size of the modules used.

1. Introduction

The evolution of digital circuits has been intensively studied to discern *generalisable* principles of design and thus to allow one automatically to produce large and efficient electronic circuits. Digital electronic circuits have been evolved intrinsically [9] and extrinsically [6, 4, 5, 18, 7]. The former is associated with an evolutionary process in which each evolved electronic circuit is built and tested on hardware, while the latter refers to circuit evolution implemented entirely in software using computer simulations.

A major problem in the evolutionary design of electronic circuits is the problem of scale. This refers to the very fast growth in the number of gates used in the target circuit as the number of inputs of the evolved logic function is increased. This results in an enormous search space that is difficult to explore even with evolutionary techniques. Another obstacle related to the problem of scale is of course the time required to calculate the fitness value of a circuit. This increases as the size of the truth table of the evolved circuit.

A possible way of tackling the problem of scale in the evolutionary design of electronic circuits has been studied in [20]. The idea was to evolve electronic circuits by using building blocks that are higher functions rather than two-input gates. It was also observed that the evolutionary design of circuits is easier when compared with the original scenario in which the building blocks are merely gates. The approach in itself is reasonable, and it has been also considered in [18]. They used binary multiplexers that are universal-logic gates to evolve arithmetic functions such as adders and multipliers. However, to identify *suitable* building blocks and thus to evolve *efficient* electronic circuits is a difficult task. For instance, the design of the three-bit multiplier circuit requires only 23 two-input gates [28], or alternatively 14 two-input gates plus 7 three-input binary multiplexers [14]. The multiplexer as an universal-logic gate requires three two-input gates and therefore the three-bit multiplier implemented by allowing multiplexers consists of 35 two-input gates. Of course the efficiency of the circuit depends entirely on one's choice of atomic gates.

The effort required to design an electronic circuit can be reduced by decomposing the circuit to sub-circuits that are easier to design, and then to consider the evolved solutions as building blocks. This relates to the concept of *automatically defined functions* studied in [11]. It was shown that in general the method produces solutions that are simpler and smaller than the solutions obtained with other techniques. A similar idea in a slightly different context has been also investigated in [26], and it has been reported that the evolutionary design of decomposed circuits is easier.

To decompose a circuit to smaller sub-circuits that are easier to evolve might be a difficult problem. This appears to be the case for the design of binary multiplier circuits. If the evolved adder circuits use the *ripple-carry adder* principle widely used in the conventional design, the evolved multipliers implement the corresponding logic operation in a very *unusual* way [14]. Perhaps suitable building blocks for the evolutionary design of digital circuits can be identified by looking at other evolved designs. The idea has been suggested in [14] and it defines the approach undertaken in this paper.

The paper studies the evolutionary design of combinational circuits, particularly the three-bit multiplier circuit, in which the basic building blocks are small sub-circuits that are modules inferred from other evolved designs. Again, the evolutionary algorithm used is Cartesian Genetic Programming [13, 17] since it allows to embed modules of any size within the genotype. The scaled evolutionary design of the three-bit multiplier is further considered as a search on a fitness landscape [8], and thus it is shown that the principles of evolving this digital circuit [29, 31, 30, 14] are scalable. To evolve digital circuits using bigger building blocks is faster, however this is also a disadvantage, since the number of gates of the evolved designs grows as the size of the modules used. It is argued however that the resulting redundancy will be less, if the building blocks used in the scaled evolution of digital circuits are defined with respect the *modularity* of evolved designs. This is supported by the results reported below.

2. Cartesian Genetic Programming

The evolutionary algorithm used in the design of digital circuits is that adopted in the framework of *Cartesian Genetic Programming* in which the genotypes are rectangular arrays rather than trees as defined in Genetic Programming [10]. The computational model has been proposed in [13, 17, 14], simplified forms of which can be traced back to earlier attempts of evolving electronic circuits [12, 18, 25, 16, 15]. The computational model has some similarities with other graph based forms of Genetic Programming such as Parallel Distributed Genetic Programming proposed by [21], and represents a dataflow graph [2].

2.1. The Evolutionary Algorithm

The algorithm deals with a population of programs that are instances of a particular program. The population consists of $1 + \lambda$ genotypes. Initially the elements of the population are chosen at random. Once the fitness values of the genotypes are evaluated a mechanism of population update is applied. The mechanism of update is implemented by truncation selection and mutation that has similarities with other evolutionary techniques such as $(1 + \lambda)$ Evolution Strategy [22, 1] and the Breeder Genetic Algorithm [19]. To update the population, the mutation operator is applied to the fittest genotype, and hence, an offspring is generated. The offspring together with the parent constitute the new population. The mutation operator is defined as the percentage of genes in a single genotype that are to be randomly mutated. The percentage usually is chosen to result in about $\mathbf{3}$ mutated genes per genotype.

2.2 The Genotype-Phenotype Mapping

To encode a program into a genotype, a genotype-phenotype mapping is defined. This is done via rectangular array of nodes that may have multiple inputs and outputs. The nodes (cells) represent atomic functions, for instance *OR*, *NOT*, *AND*, etc. In general the array consists of $n \times m$ nodes, n_I inputs, and n_O outputs. The inputs of the array are the inputs of the represented phenotype that implements the target function. Alternatively, the outputs of the array represent the outputs of the phenotype. The inputs and the outputs of the array together with the inputs and the outputs of the nodes are indexed by integers and thus the routing of the array is defined.

The routing is completely defined by the internal connectivity and the output connectivity of the array. The internal connectivity is specified by the connections between the array cells. The inputs of each cell are only allowed to be inputs of the array or outputs of the cells with lower column numbers. The internal connectivity is also dependent on a *levels-back* parameter that determines the array inputs and cells to which a cell or an array output can be connected. The levels-back parameter is defined with respect the layout of the array. Let L denote the levels-back parameter. Then cells can be connected to cells from L preceding columns. If the number of preceding columns of a cell is less than L then the cell can also be connected to the inputs of the array.

The array output connectivity is defined in a similar way. The output connections of the array are allowed to be outputs of cells or array inputs. Again, this is dependent on the neighbourhood defined by the levels-back parameter. An illustration of the array of nodes is given in Figure 1.

The genotype is a linear string of integers and it consists of two different types of genes that are responsible for the functionality and the routing of the evolved array of cells. Hence, the genotype is defined by four parameters of the array: the number of allowed atomic functions, the number of rows, the number of columns, and the levels-back. The first parameter defines the functionality of cells, while the lat-

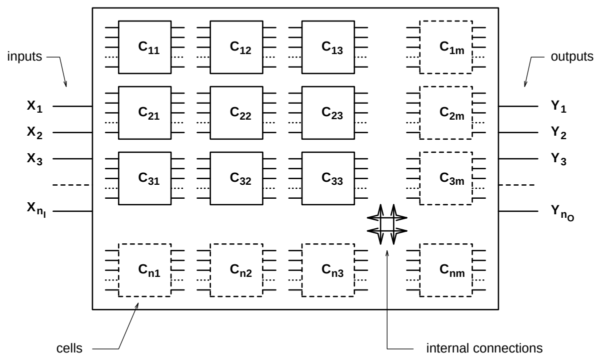


Figure 1. The phenotype is encoded within a genotype by a rectangular array of cells.

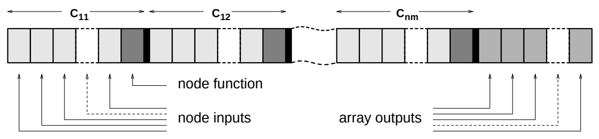


Figure 2. The genotype with respect the array of cells (nodes) from Figure 1.

ter three parameters determine the layout and the routing of the array. Note that the number of inputs and outputs of the array are specified by the objective function. The node functions are represented by letters associated with the allowed cell functionality. The connections are defined by indexes that are assigned to all components of the array, particularly the array inputs and the node outputs. Suppose the array as given in Figure 1 consists of cells each of which is a function with p inputs and q outputs. Then the k^{th} array input, X_k, is labelled with $k-1$ for $1 \leq k \leq n_I$, while the k^{th} output of cell $\mathbf{c}_{ij}$ is labelled with an integer given by the following expression

$$n_I + (j-1)qn + (i-1)q + k - 1 \qquad (1)$$

for $1 \leq i \leq n$, $1 \leq j \leq m$, and $1 \leq k \leq q$. The genotype consists of groups of $p+1$ integers that encode the cells of the array, followed by a sequence of integers that represent the indexes of the cell outputs connected to the outputs of the array. The first p values of each group are the indexes of the node outputs to which the inputs of the encoded cell are connected. The last integer of the group represents the function of the node. The genotype representation is illustrated in Figure 2. The genes are depicted with filled rectangles that differ from each other in fill intensity. The darkest rectangles represent the genes responsible for the functionality of the logic cells, the less dark are the genes for the array output connections, and the least dark are the genes for the internal connectivity.

3. Evolving Digital Circuits and Modules

The genotype-phenotype mapping allows to encode a digital combinational circuit into a genotype and thus to evolve novel and efficient designs. Basically, the circuit is treated as a graph based computational model in which the nodes are three-input logic cells each of which represents either a two-input logic gate or an universal-logic gate (2-1 binary multiplexer). Thus the genotype consists of groups of 4 integers that encode the cells of the array, followed by a sequence of integers that represent the indexes of the cells connected to the outputs of the array. The first 3 values of each group are the indexes of the cells to which the inputs of the encoded cell are connected, while the last integer of the group represents the logic gate. The fitness of a genotype is the number of correct output bits.

The allowed logic gates used in the evolutionary design of three-bit binary multiplier circuits are *AND*, *AND* with one input inverted, and *XOR*. These are referred to as gates 6, 7, and 10, respectively. The most efficient evolved design consists of 23 two-input gates that is 23.3% more efficient than the most efficient conventional design. Many 23 gates solutions were found, and it was revealed that they are very similar in construction. One of them is shown in Figure 3.

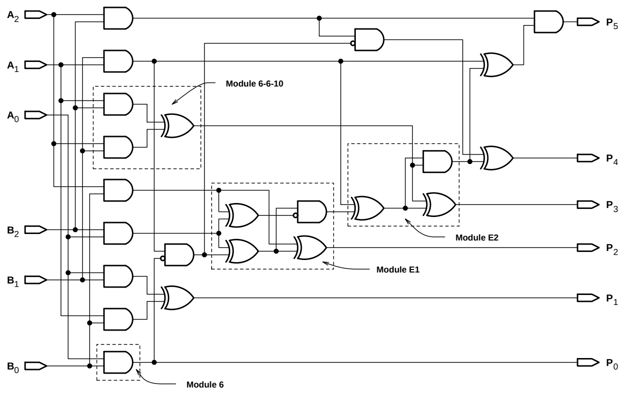


Figure 3. An evolved three-bit multiplier circuit. The circuit consists of 23 two-input gates: *AND*, *XOR*, and *AND* with one input inverted.

A simple observation of the evolved 23 gates solutions revealed that there are sub-circuits that are *typical* for the designs (see also [14, 28]). These are referred to as modules and they can be classified in three groups. The first group represents the two-input gates. The second group consists of all possible sub-circuits of three gates in which the output of the module is taken from a gate with inputs connected to the other two gates. An example of such is module $6-6-10$ shown in Figure 3. The last group consist of two types of sub-circuits. These are referred to as modules E_1 and E_2 (Figure 3) and they represent logic functions that have several different circuit implementations. The instances of modules E_1 and E_2 encountered so far are shown in Figures 4 and 5, respectively. The outputs of module E_1 are

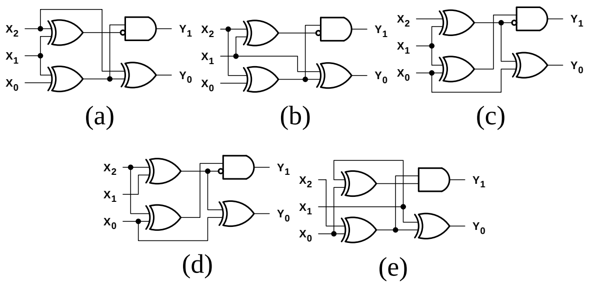


Figure 4. Logically equivalent sub-circuits for module E_1.

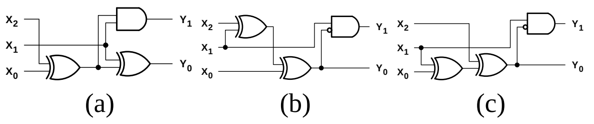


Figure 5. Logically equivalent sub-circuits for module E_2.

given by the following logic expressions

$$Y_0 = X_0 \oplus X_1 \oplus X_2 \tag{2}$$

and

$$\begin{aligned} &(1) \quad Y_1 = (X_0 \oplus X_1) \cdot \overline{X_1 \oplus X_2} \\ &(2) \quad Y_1 = (X_0 \oplus X_2) \cdot \overline{X_1 \oplus X_2} \\ &(3) \quad Y_1 = (X_0 \oplus X_1) \cdot (X_0 \oplus X_2) \end{aligned} \tag{3}$$

where "$\oplus$" is *XOR*, "$\cdot$" is *AND*, and the bar is inversion. Note that the logic functions of output Y_1 are different, however embodied in the three-bit multiplier they represent logically equivalent sub-circuits. The outputs of module E_2 are given by equation 2 and

$$\begin{aligned} &(1) \quad Y_1 = X_1 \cdot (X_0 \oplus X_2) \\ &(2) \quad Y_1 = X_1 \cdot \overline{X_0 \oplus X_1 \oplus X_2}. \end{aligned} \tag{4}$$

Once again, output Y_1 is implemented in two different ways. The logic equivalency is to be expected since this is one way to avoid the destructive effect of the non-beneficial mutations during the evolutionary search. This is an advantage of the evolved designs since they are characterised with *fault tolerance*.

4. Embedding Modules within the Genotype

There are two main obstacles that one may encounter when attempting to encode the modules within the genotype and thus to evolve with bigger building blocks. Firstly, the modules have different numbers of inputs and outputs. Secondly, the evolved designs are characterised with a very high degree of reuse. The latter refers to cases in which inputs of two or more gates are connected to one preceding gate. A possible solution to these problems is to define the building blocks with respect to the modules with higher numbers of inputs and outputs. In addition the number of outputs of each building block must be high enough to allow the reuse of gates from inside the block. This is illustrated in Figure 6. The figure represents three examples

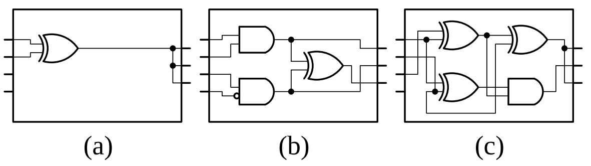


Figure 6. The building blocks and the three types of modules.

of building blocks each of which embodies a module from a particular class. Thus the block in Figure 6a represents module 10 that is a two-input gate, the block in Figure 6b is module $6-7-10$ since it consists of gates 6, 7, and 10, and lastly, the block in Figure 6c is module E_1 that is an instance from the third group of sub-circuits (Figure 4). Note that the block shown in Figure 6b allows to reuse the two-input gates from inside the module.

The building blocks are four-input cells that can be arranged in a rectangular array, and thus, by using the genotype-phenotype mapping described in section 2 to construct the new genotype. The allowed atomic functions used in this approach are given in Table 1. The mutation results in three randomly mutated genes per genotype. Again the fitness of a genotype is the number of correct output bits. This is scaled in the interval $[0, 1]$. In the experiments below, the array consists of 1×14 cells, and the levels-back parameter is set to 14, and the target circuit is the three-bit multiplier.

Letter	Module
6	6
7	7
10	10
21	$6-6-7$
22	$6-6-10$
24	$6-10-10$
25	$6-7-10$
31	E_1
32	E_2

Table 1. The atomic logic functions (modules) used.

5. Landscape Structure and Search

The resulting fitness landscape is a product of three subspaces with different characteristics. These are the functionality, internal connectivity, and output connectivity landscapes. Hence the underlying graph is the Generalised Hamming hypercube, a product of the three configuration spaces. The landscape is statistically non-isotropic. This was revealed by measuring the autocorrelation functions from different starting points. The landscape structure is studied on $1,000$ random walks with length $100,000$ per configuration space. The starting points were chosen randomly with fitness values uniformly distributed in the interval $[0.5 : 1]$. Thus $1,000$ time series were generated for each family of landscapes. It is important to note that each random walk was performed with respect the studied subspace.

5.1. Correlation Analysis

The correlation structure of the landscape is investigated by measuring the autocorrelation functions of time series sampled on random walks [32, 23]. The autocorrelation function of time series $\{f_t\}_{t=0}^{n}$ is given by

$$\rho(s) = \frac{E[f_t f_{t+s}] - E[f_t]E[f_{t+s}]}{V[f_t]} \qquad (5)$$

where $E[f_t]$ and $V[f_t]$ are the expectation and variance, respectively. The measured autocorrelations are shown in Figure 7. The figure represents the average autocorrelations of functionality, internal connectivity, and output connectivity landscapes that resulted from onepoint mutation, and the corresponding standard deviations. The results are depicted with scaled Hamming distance, d/n, in the same manner as was done in [24].

The figure reveals that the output connectivity landscapes are smoother and the internal connectivity landscapes are more rugged than the functionality landscapes. This implies that the structure of the investigated landscapes is similar to the landscape structure studied in the original scenario [29]. When comparing the plots of the autocorrelation functions for the two cases, non-scaled and scaled evolutionary designs, one may notice that the plots for the internal and output connectivity landscapes are virtually the same. However, this is not the case for the functionality landscapes. In the new scenario investigated in this paper, the functionality landscapes are smoother. This implies that by using building blocks inferred from evolved designs one may *improve* the evolutionary search.

The plots of the standard deviations reveal how the functionality, internal connectivity, and output connectivity landscapes are related. Again the result implies certain similarities between the two scenarios. The figure reveals that

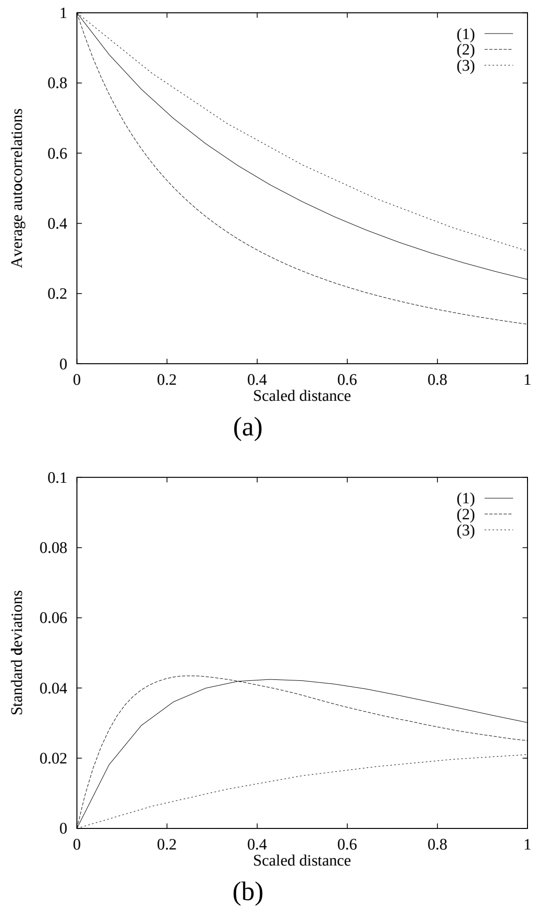
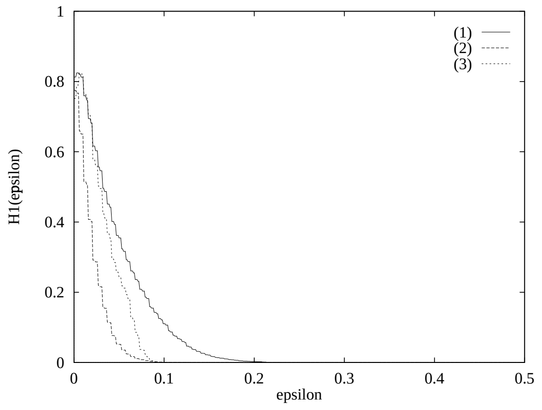


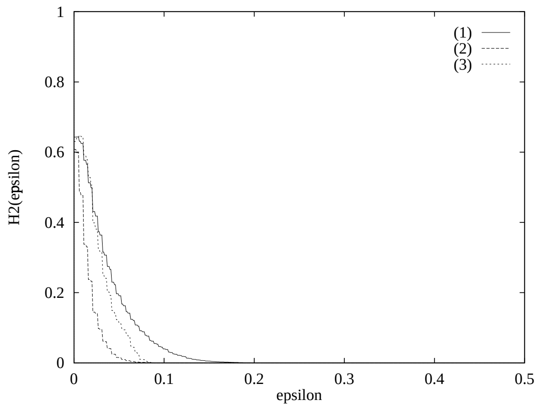
Figure 7. Correlation structure: (a) average autocorrelations, and (b) standard deviations with scaled distance of (1) functionality, (2) internal connectivity, and (3) output connectivity landscapes.

the smoothest subspace is less dependent on the structure of the other two landscapes. However, the standard deviations for the most rugged subspace (that is the internal connectivity landscape) differ from the corresponding result for the non-scaled evolutionary design. The amplitude spectra of the landscapes were also measured and it was found that the number of classes of the amplitude spectra of internal connectivity landscapes is less than the number of classes identified for the functionality landscapes. However this divergence from the original scenario is slight and it might be caused by the changes in the array layout and the levels-back parameter.

5.2. Information Structure



(a)



(b)

Figure 8. Information structure: (a) information function $H_1(\varepsilon)$, and (b) information function $H_2(\varepsilon)$ of (1) functionality, (2) internal connectivity, and (3) output connectivity landscapes.

The information structure is studied by measuring the information characteristics on three random walks, a random walk per configuration space. It has been shown that the information analysis can reveal aspects smoothness, ruggedness, and neutrality of landscapes [27, 14], and it is implemented by measuring the information functions $H_1(\varepsilon)$ and $H_2(\varepsilon)$ associated with the first and second entropic measures respectively (see [14]). The measured functions are depicted in Figure 8.

The figure reveals that the structure of the fitness landscapes for both scenarios, non-scaled and scaled evolutionary designs, have great similarities. The landscapes are characterised with neutrality and sharply differentiated plateaus. Again, the neutrality prevails over the landscape smoothness and ruggedness only for internal connectivity subspaces. The output connectivity landscapes are characterised with less neutrality than the functionality and internal connectivity landscapes.

6. Evolvability and Efficient Designs

Thus far the results imply that in general the principles of evolving digital circuits are scalable. Therefore to evolve the three-bit multiplier is expected to be easier since the building blocks are bigger [20]. This is revealed by comparing typical evolutionary runs for the two scenarios of evolving the three-bit multiplier. The scaled evolution is implemented on an array of 1×14 cells and levels-back set to 14. The allowed atomic functions are these listed in Table 1. The non-scaled evolution that is the original scenario is implemented on an array of 1×30 cells and levels-back equal to 30. The allowed functions are 6, 7, 10, and 16 where 16 is a binary multiplexer. The multiplexer is the universal-logic gate $f_L(a, b, c) = a \cdot \overline{c} + b \cdot c$ as given in [3]. The first evolutionary run gave a solution in generation $397,605$ while the second in generation $5,629,540$. The best fitness plots of the runs are depicted in Figure 9. In terms of number of generations required to evolve the circuit, the scaled evolutionary design appeared to be approximately 14 times more efficient.

The improvement is significant even for scaled evolution. Note that the building blocks are slightly bigger than the cells in the original scenario where binary multiplexers were used. These have three inputs and one output, while the cells in the scaled scenario have four inputs and three outputs. Possible reasons for this significant improvement are the following: firstly, the building blocks are chosen from evolved designs, and secondly, they allow the reuse of gates from inside the modules. These characteristics of the building blocks define an important advantage in the new scaled scenario. This concerns the issue of the efficiency of the evolved digital circuit in terms of number of two-input gates used. The most efficient three-bit multiplier evolved

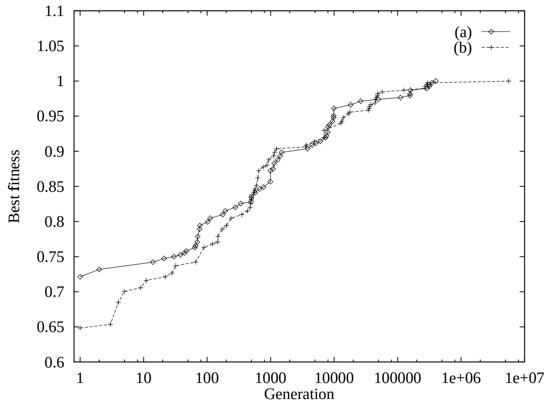


Figure 9. Typical evolutionary runs for the two scenarios: (a) scaled, and (b) non-scaled evolutionary designs of the three-bit multiplier circuit.

with binary multiplexers consists of 14 two-inputs plus 7 multiplexers that results in exactly 35 two-input gates [14]. However, the design obtained in the scaled evolutionary run consists of 27 two-input gates (Figure 11). This implies that the modules chosen in this approach allow one to evolve more efficient designs for the three-bit multiplier.

Many evolutionary runs were performed. The most efficient design obtained thus far was the circuit given in Figure 11. Although the circuit is more efficient than the conventional design, it is still far away from the optimal solution for the three-bit multiplier evolved in [28] (Figure 3). A simple study of the structure of the evolved design leads to an astonishing conclusion. Evolution builds the three-bit multiplier by assembling the building blocks again, disregarding the fact that these are already available! This is illustrated in Figure 10. The depicted circuit is taken from the

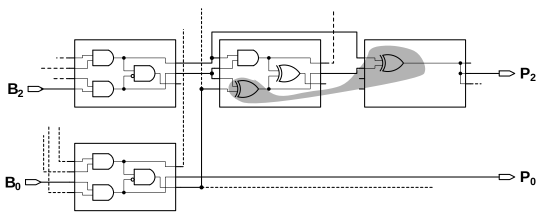


Figure 10. A sub-circuit taken from the evolved three-bit multiplier shown in Figure 11.

evolved three-bit multiplier shown in Figure 11. The shadowed gates illustrate how evolution builds a half of module E_1 (Figure 4). Of course this results in the usage of some extra gates for implementing the other half of the module. This can also be observed in the remainder of the circuit. For instance, module E_1 is used in the implementation of module E_2. The *overlapping* and the *strange reuse* of modules are inevitable in the scaled design of digital circuits and they can only be avoided by identifying modules that are suitable for evolution. Such modules can be discerned from evolved circuits by studying the principles of building the logic operation of the circuits.

7. Conclusions

Large digital circuits are difficult to evolve. A possible way to solve this problem of scale is to use larger circuits as basic building blocks and thus to evolve bigger circuits. The main advantage of this approach is that digital circuits with scaled building blocks are easier to evolve. The reason is of course the bigger size of blocks used. The paper showed that the structure of the resulting fitness landscapes is similar to the landscape structure defined in the evolutionary design with two-input gates. This implies that the principles of evolving digital circuits are scalable, and therefore, the effort required to evolve circuits of n blocks is invariant to the size of the blocks.

The major impediment in scaling the evolutionary design of digital circuits is the understanding how to define building blocks that are suitable for evolution. This problem was easily solved in the case of binary adder circuits [14]. It was found that a large efficient adder can be designed by using adder of smaller size as building blocks. However, this does not appear to be the case for the binary multiplier circuits, particularly the three-bit multiplier.

In the conventional design the multiplier can be easily built by applying the cellular multiplier principle of constructing multiplication by binary adders. However, the evolved solutions for this combinational circuit differ from the conventional design. The multiplier circuit designed by evolution does not use adders and multipliers of smaller size. Therefore, to define suitable modules for the design of this particular circuit is a very difficult task. Perhaps the efficient multiplier is a basic computational unit that is impossible to represent by smaller blocks. The question that arises here is how to scale the evolutionary design of this arithmetic circuit? The answer is probably inside the circuit, in the *evolutionary principle* of multiplication. In itself the problem does not concern only the multiplier circuit design. There are probably other arithmetic and logic functions for which the most efficient designs cannot be decomposed to sub-circuits. These are all questions for future research.

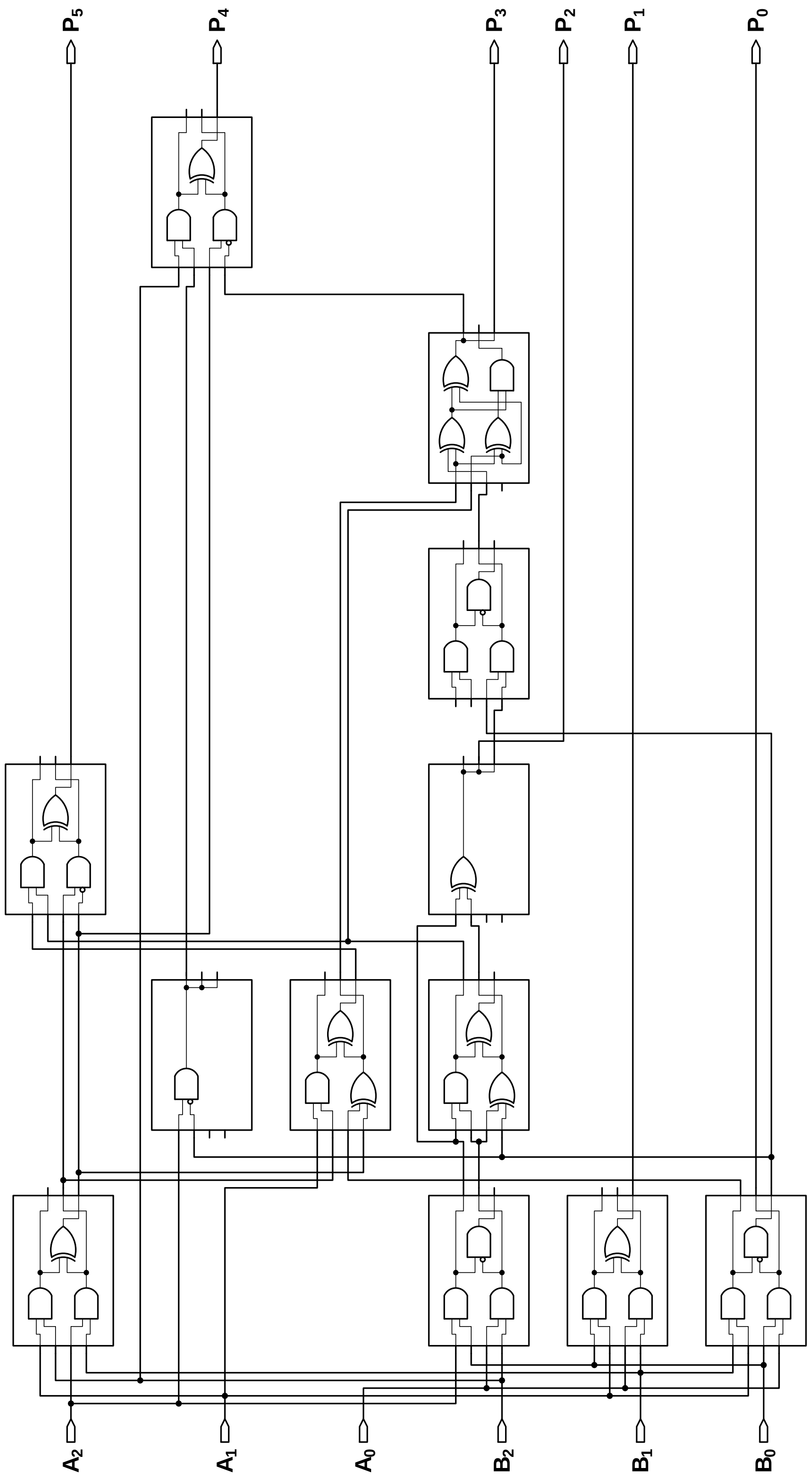


Figure 11. A three-bit multiplier circuit obtained by evolving *evolved* modules.

References

[1] T. Bäck, F. Hoffmeister, and H. P. Schwefel. A survey of evolutionary strategies. In R. Belew and L. Booker, editors, *Proceedings of the 4th International Conference on Genetic Algorithms*, pages 2–9, San Francisco, CA, 1991. Morgan Kaufmann.

[2] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming: An Introduction.* Morgan Kaufmann, San Francisco, CA, 1998.

[3] X. Chen and S. L. Hurst. A comparison of universal-logic-module realizations and their application in the synthesis of combinatorial and sequential logic networks. *IEEE Transactions on Computers*, C-31(2):140–147, 1982.

[4] H. Hemmi, T. Hikage, and K. Shimohara. Adam: A hardware evolutionary system. In *Proceedings of the 1st International Conference on Evolutionary Computation*, volume 1, pages 193–196, Piscataway, NJ, 1994. IEEE press.

[5] H. Hemmi, J. Mizoguchi, and K. Shimohara. Development and evolution of hardware behaviours. In R. Brooks and P. Maes, editors, *Artificial Life IV: Proceedings of the 4th International Workshop on the Synthesis and Simulation of Living Systems*, pages 371–376, Cambridge, MA, 1994. MIT Press.

[6] T. Higuchi, T. Niwa, T. Tanaka, H. Iba, H. de Garis, and T. Furuya. Evolving hardware with genetic learning: A first step towards building a darwin machine. In J.-A. Meyer, H. L. Roitblat, and W. Stewart, editors, *From Animals to Animats II: Proceedings of the 2nd International Conference on Simulation of Adaptive Behaviour*, pages 417–424, Cambridge, MA, 1993. MIT Press.

[7] H. Iba, M. Iwata, and T. Higuchi. Machine learning approach to gate-level evolvable hardware. In T. Higuchi and M. Iwata, editors, *Proceedings of the 1st International Conference on Evolvable Systems: From Biology to Hardware*, volume 1259 of *Lecture Notes in Computer Science*, pages 327–343, Heidelberg, 1997. Springer-Verlag.

[8] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search*. PhD thesis, University of New Mexico, Albuquergue, NM, 1995.

[9] I. Kajitani, T. Hushino, D. Nishikawa, H. Yokoi, S. Nakaya, T. Yamauchi, T. Inuo, N. Kajihara, M. Iwata, D. Keymeulen, and T. Higuchi. A gate-level ehw chip: Implementing ga operations and reconfigurable hardware on a single lsi. In M. Sipper, D. Mange, and A. Pérez-Uribe, editors, *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, volume 1478 of *Lecture Notes in Computer Science*, pages 1–12, Heidelberg, 1998. Springer-Verlag.

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

[11] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, 1994.

[12] S. J. Louis. *Genetic Algorithms as a Computational Tool for Design.* PhD thesis, Department of Computer Science, Indiana University, Indiana, 1993.

[13] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the 1st Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, San Francisco, CA, 1999. Morgan Kaufmann.

[14] J. F. Miller, D. Job, and V. K. Vassilev. Principles in the evolutionary design of digital circuits. *Journal of Genetic Programming and Evolvable Machines*, 1(1/2,3):8–35, in press, 2000.

[15] J. F. Miller and P. Thomson. Aspects of digital evolution: Evolvability and architecture. In A. E. Eiben, T. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature V*, volume 1498 of *Lecture Notes in Computer Science*, pages 927–936, Berlin, 1998. Springer.

[16] J. F. Miller and P. Thomson. Aspects of digital evolution: Geometry and learning. In M. Sipper, D. Mange, and A. Pérez-Uribe, editors, *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, volume 1478 of *Lecture Notes in Computer Science*, pages 25–35, Heidelberg, 1998. Springer-Verlag.

[17] 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, *Proceedings of the 3rd European Conference on Genetic Programming*, volume 1802 of *Lecture Notes in Computer Science*, pages 121–132, Berlin, 2000. Springer-Verlag.

[18] J. F. Miller, P. Thomson, and T. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In D. Quagliarella, J. Periaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pages 105–131. Wiley, Chechester, UK, 1997.

[19] H. Mühlenbein and D. Schlierkamp-Voosen. The science of breeding and its application to the breeder genetic algorithm (bga). *Evolutionary Computation*, 1(4):335–360, 1993.

[20] M. Murakawa, S. Yoshizawa, I. Kajitani, T. Furuya, M. Iwata, and T. Higuchi. Hardware evolution at functional level. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature IV*, volume 1141 of *Lecture Notes in Computer Science*, pages 62–71, Berlin, 1996. Springer.

[21] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In T. Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 346–353, San Francisco, CA, 1997. Morgan Kaufmann.

[22] H.-P. Schwefel. *Numerical Optimization of Computer Models.* John Wiley & Sons, Chichester, UK, 1981.

[23] P. F. Stadler. Towards theory of landscapes. In R. Lopéz-Pena, R. Capovilla, R. García-Pelayo, H. Waelbroeck, and F. Zertuche, editors, *Complex Systems and Binary Networks*, pages 77–163. Springer-Verlag, Berlin, 1995.

[24] P. F. Stadler and W. Grünter. Anisotropy in fitness landscapes. *Journal of Theoretical Biology*, 165:373–388, 1993.

[25] A. Thompson. *Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution.* Springer-Verlag, London, 1998.

[26] J. Torresen. A divide-and-conquer approach to evolvable hardware. In M. Sipper, D. Mange, and A. Pérez-Uribe, editors, *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, volume 1478 of *Lecture Notes in Computer Science*, pages 57–65, Heidelberg, 1998. Springer-Verlag.

[27] V. K. Vassilev, T. C. Fogarty, and J. F. Miller. Information characteristics and the structure of landscapes. *Evolutionary Computation*, 8(1):31–60, 2000.

[28] V. K. Vassilev and J. F. Miller. Embedding landscape neutrality to build a *bridge* from the conventional to a more efficient three-bit multiplier circuit. In *Proceedings of the 2nd Genetic and Evolutionary Computation Conference*, San Francisco, CA, 2000. Morgan Kaufmann. To appear (available via http://www.dcs.napier.ac.uk/∼vesselin/).

[29] V. K. Vassilev, J. F. Miller, and T. C. Fogarty. Digital circuit evolution and fitness landscapes. In *Proceedings of the Congress on Evolutionary Computation*, volume 2, pages 1299–1306, Piscataway, NJ, 1999. IEEE Press.

[30] V. K. Vassilev, J. F. Miller, and T. C. Fogarty. Digital circuit evolution: The ruggedness and neutrality of two-bit multiplier landscapes. In D. M. Harvey, editor, *Evolutionary Hardware Systems*, pages 6/1–6/4, London, 1999. IEE Press.

[31] V. K. Vassilev, J. F. Miller, and T. C. Fogarty. On the nature of two-bit multiplier landscapes. In A. Stoica, D. Keymeulen, and J. Lohn, editors, *Proceedings of the 1st NASA/DoD Workshop on Evolvable Hardware*, pages 36–45, Los Alamitos, CA, 1999. IEEE Computer Society.

[32] E. D. Weinberger. Correlated and uncorrelated fitness landscapes and how to tell the difference. *Biological Cybernetics*, 63:325–336, 1990.