# Improving the Evolvability of Digital Multipliers using Embedded Cartesian Genetic Programming and Product Reduction

James Alfred Walker

Department of Electronics
University of York
Heslington, York
YO10 5DD, UK
jaw500@ohm.york.ac.uk

Julian Francis Miller

Department of Electronics
University of York
Heslington, York
YO10 5DD, UK
jfm@ohm.york.ac.uk

**Abstract.** Embedded Cartesian Genetic Programming (ECGP) is a form of Genetic Programming based on an acyclic directed graph representation. In this paper we investigate the use of ECGP together with a technique called Product Reduction (PR) to reduce the time required to evolve a digital multiplier. The results are compared with Cartesian Genetic Programming (CGP) with and without PR and show that ECGP improves evolvability and also that PR improves the performance of both techniques by up to eight times on the digital multiplier problems tested.

## 1 Introduction

The evolution of digital multipliers has proved to be very difficult for evolutionary techniques (particularly when the number of bits in the multiplicands is greater than three) [4][7][11][12][13]. Cartesian Genetic Programming (CGP) [5][6] is one technique that has been used to attack such problems. Even though CGP does not have the equivalent of Automatically Defined Functions (ADFs) it was empirically demonstrated to be more computationally efficient than Genetic Programming (GP) [3] with Automatically Defined Functions (ADF's) on the even parity and 2-bit multiplier problems [5]. Embedded Cartesian Genetic Programming (ECGP) is a development of CGP that allows the construction and evolution of modules that can be called from the main CGP code and has been shown to perform better than standard CGP on a series of parity problems [14]. In this paper we apply ECGP to the multiplier problem. We also introduce a new approach called Product Reduction (PR), which is designed to make evolving digital multipliers easier.

The plan for the paper is as follows: Section 2 is an overview of related work. In section 3 we describe ECGP and compare it with CGP before describing PR in section 4. The details of our experiments are shown in section 5 followed by the results and comparisons for all three experiments in section 6. Section 7 gives conclusions and some suggestions for future work.

## 2 Module Acquisition and Automatically Defined Functions

Module acquisition (MA) [1] adds two operators to the evolutionary process, *compress* that selects a section of the genotype to make it immune to manipulation from operators (the module) and *expand* which decompresses a module in the genotype therefore allowing this section of the genotype to be manipulated once more. The fitness of a genotype is unaffected by these operators. MA allows the possibility of having modules within modules. These techniques have been shown to decrease the time taken to find a solution. Rosca's method of Adaptive Representation through Learning (ARL) [8] also extracted program segments that were encapsulated and used to augment the GP function set. However, recently Dessi et al [2] showed that random selection of program sub-code for re-use is more effective than other Rosca's method across a range of problems. Once the contents of modules are themselves allowed to evolve (as in ECGP) they become a form of ADF, however in contrast to Koza's form of ADFs [3] and Spector's Automatically Defined Macros [9], there is no explicit specification of the number or internal structure of such modules. This freedom does exist in Spector's PushGP [10].

## 3 Embedded Cartesian Genetic Programming (ECGP)

### 3.1 Representation

ECGP and CGP share the same structure and represent a program as a directed graph (that for feed-forward functions is acyclic). The genotype is a list of integers that encode the connections and functions of each node of the directed graph. CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. However, later work in CGP always chose the number of rows to be one, thus giving a one-dimensional topology. This is always used in ECGP. In CGP, the genotype is a fixed length representation (in terms of genes) in which the number of nodes in the program (phenotype) can vary but is bounded. In ECGP the genotype is a variable length representation (in terms of genes and nodes) in which the number of nodes and genes in the graph is bounded. The variable number of nodes in the ECGP genotype is the result of the compression and expansion of modules and the variable number of genes (which allows each node to have a variable number of inputs) is a result of the re-use of modules and some of the module mutation operators, which can change the number of inputs of a node. In Fig. 1 an example of the differences between a CGP and an ECGP genotype are shown. Despite these differences, both CGP and ECGP are initialized with a CGP style genotype. This means that all of the initial genotypes in the population have the same number of nodes and genes and every node represents a primitive function (i.e. no modules are present). Each of the nodes consists of two parts: a node header and a node body. The node header encodes the primitive function or module (by their unique identifier) that the node represents and the type of the node (type I or type II) if the node represents a module (the concept of module type is explained in section 3.4). The node body en-

codes the inputs of the node. Each input is encoded by two integers: one represents the index of the node or program input (terminal) in the genotype and the other represents the output of the node (note nodes can have multiple outputs) – see Fig. 2. The number of inputs and outputs that a node has is dictated by the arity of its function.

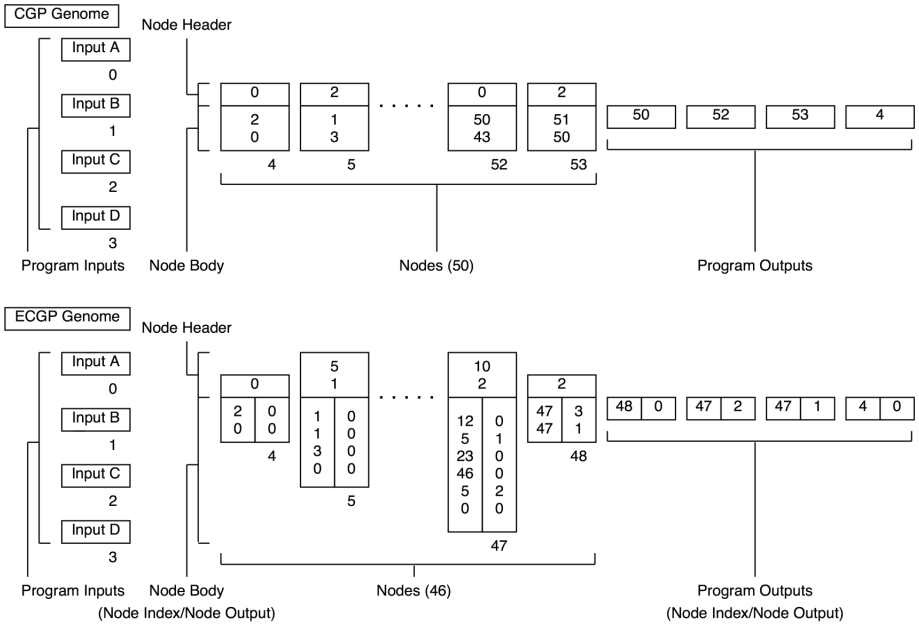


**Fig. 1.** Examples of evolved CGP and ECGP genotypes for a 2-bit digital multiplier (4 inputs, 4 outputs). Both genotypes were initialized with 50 nodes (150 genes). The top or only number in the node headers represents the function; the remaining number (where present) is the node type. The node body represents the node inputs, which in ECGP are split into two parts: the node index in the genotype and the points from which the node outputs are taken. The node index is underneath each node.

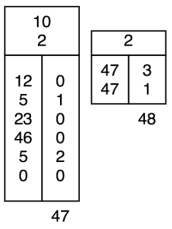

**Fig. 2.** A fragment of the ECGP genotype from Fig.1. Node 47 with function 10 is of type II (see later) and is a module with 6 inputs connected to node indexes 12(0), 4(1), 23(0), 46(0), 4(2), 0(0). The particular outputs of the nodes are in brackets. Node 48 with function 2 is a primitive function whose inputs are both taken from node 47. The first input comes from output 3 and the second comes from output 1.

The nodes take their inputs in a feed forward manner from either the output of a previous node or from a program inputs (terminals). The program inputs are numbered from 0 to $n$-1 where $n$ is the number of program inputs. The nodes in the genotype are also numbered sequentially starting from $n$ to $n+m$-1 where $m$ is the user-determined upper bound of the number of nodes. If the problem requires $k$ program outputs then $k$ integers are added to the end of the genotype, each one representing a pointer to the output of a node in the graph where the program output is taken from. These $k$ integers are initially set as pointers to the outputs of the last $k$ nodes in the genotype. Fig. 3 shows an ECGP genotype and how it is decoded (a 2-bit digital multiplier circuit).

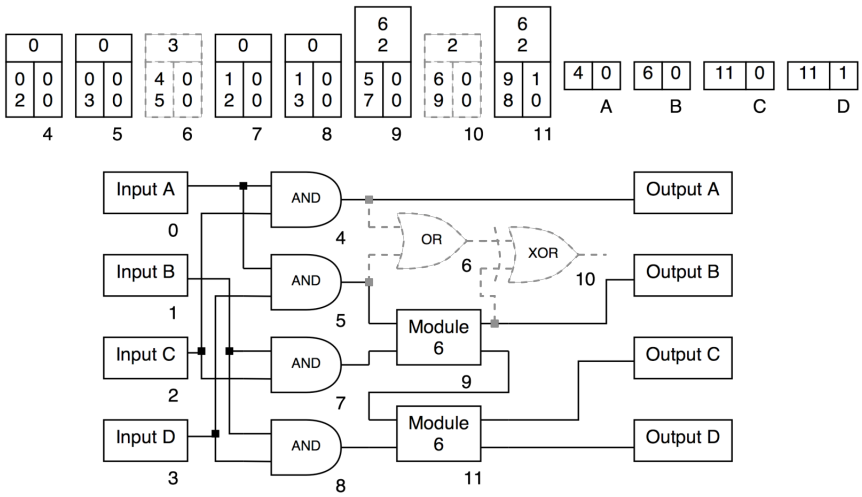


**Fig. 3.** An ECGP genotype and corresponding phenotype for a 2–bit digital multiplier circuit. Module 6 represents a possible structure for a half adder constructed from the function set. The inactive areas of the genotype and phenotype are shown in grey dashes.

Although each node must have a function and a set of inputs for that function, the outputs of a node do not have to be connected. This is shown in Fig. 3, where the output of node 10 is not used. This allows areas of the genotype to be inactive (nodes 6 and 10, shown in grey dashed lines), leading to a neutral effect on genotype fitness (neutrality). When point mutations are carried out on genes representing connections (the mutation is constrained to respect the directed and acyclic nature of the graphs) these inactive genes can be activated or active genes can be made inactive.

### 3.2 Evolutionary Strategy

We have used a 1+4 evolutionary strategy defined below:

1. Randomly generate an initial population of 5 genotypes and select the fittest.
2. Carry out point-wise mutation on the winning parent to generate 4 offspring.
3. Construct a new generation with the winner and its offspring.

4. Select a winner from the current population using the following rules:
– If any offspring has a better fitness; the best becomes the winner.
– Otherwise, an offspring with the same fitness as the best is randomly selected.
– Otherwise, the parent remains as the winner.
5. Go to step 2 unless the maximum number of generations is reached or a solution is found.

### 3.3 Module Representation

A module is represented as a bounded variable length genotype that has the same characteristics of a standard CGP genotype. The module genotype consists of a list of integers and is split into two parts: the module header and the module body. The module header contains four integers and stores information about the module. Each of the four integers encodes the module identifier, the number of module inputs, the number of nodes contained in the module and the number of module outputs respectively. The module body encodes the connections and functions of the nodes contained in the module and the module outputs (similar to program outputs) in the same way as any standard CGP genotype. An example of a module genotype showing the separate components is shown in Fig. 4.

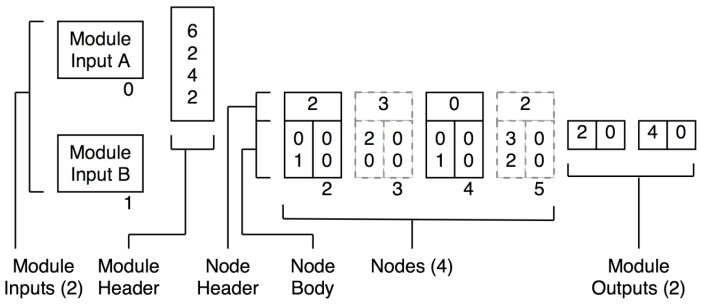


**Fig. 4.** An example of a module genotype. The four numbers in the module header represent the module identifier, the number of inputs, the number of nodes and the number of outputs of the module respectively. The nodes are represented the same as in ECGP. The module outputs represent which nodes the module takes its outputs from.

The size of a module genotype is determined by the number of nodes and module outputs that it encodes. The number of nodes encoded in the module genotype is bounded between a minimum limit of two (any fewer and it would either be an empty module or a primitive function) and a maximum limit that is set by the user. Likewise the number of module outputs encoded in the module genotype is also bounded between a minimum limit of one (otherwise there would be no way to connect to the module and access its result to the given inputs) and a maximum of n module outputs, where n is equal to the number of nodes contained in the module (one module output per node). The number of module inputs that a module is allowed to have is also restricted between a minimum of two and a maximum of 2n module inputs, where n is

equal to the number of nodes contained in the module. However, the number of module inputs allowed does not affect the size of the module genotype, as they are not encoded in the module genotype. In its current form, ECGP only allows modules to contain nodes representing primitive functions rather than nodes representing other modules. An example is given in Fig. 5.

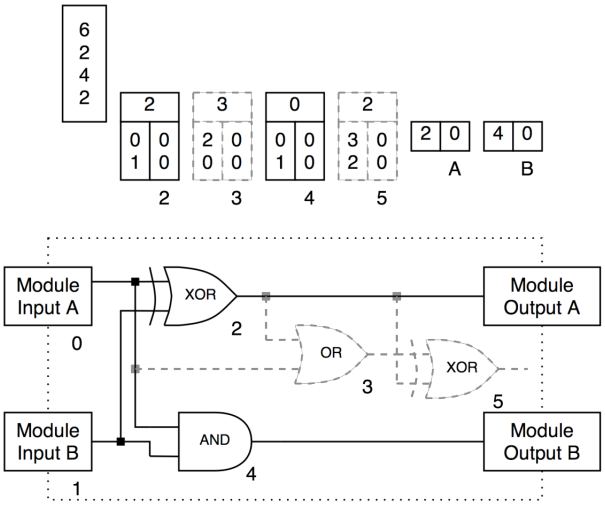


**Fig. 5.** The genotype and corresponding phenotype of a module representing a half adder. The inactive areas of the genotype and phenotype are shown in grey dashes. The dotted box represents the edges of the module.

Once a module is created, the module genotype is stored in the module list, which is an extension of the primitive function list. This allows any node in the genotype of an individual to be mutated into any module or primitive function present in either of these lists for that generation. The module list is dynamic and has no restrictions on its maximum size and is updated every generation when the fittest individual (chosen in accordance with the evolutionary strategy used in Section 3.2) in the generation is promoted to the next generation (i.e. the next generation inherits the module list of the fittest individual in the previous generation). This creates a regulatory control of the module list so that bloat never occurs.

The nodes contained inside the module are not necessarily connected and are immune from the main genotype point mutation operator. However, the module itself is allowed to be mutated by the module mutation operators (including a module point operator see section 3.4).

### 3.4 Operators

ECGP extends CGP by allowing the use of dynamic acquisition, evolution and the reuse of modules. This is achieved through extra mutation operators, which are used in conjunction with the genotype point mutation of CGP.

The compress operator constructs modules by selecting two random points in the genotype (in accordance with the rules for the module size restrictions) and encapsulates all the nodes between these two points into a new module, which is encoded into a module genotype as described earlier. Note, that if there any modules between the two selected points the compress operator does not take place (this is because at present we do not allow modules within modules). The number of module inputs that a module is initialized with is determined by the number of connections between the inputs of the nodes that are going to be encapsulated into a module and the outputs of any previous nodes or program inputs (terminals) in the genotype when the module is created. Likewise, the number of module outputs possessed by a module is determined by the number of connections between the inputs of the latter nodes in the genotype and the outputs of the nodes that are going to be encapsulated in the module, when it is created. Any module created by the compress operator is represented in the genotype of an individual as a type I node. The node header (i.e. the primitive function or module that the node represents) in any type I node is immune from the genotype point mutation operator therefore allowing the type I node to remain in the genotype of an individual until it is removed by the expand operator (see Table 1).

The expand operator destroys a type I node by replacing it in the genotype of an individual with the nodes contained in the module that the type I node represented. The inputs of all of the latter nodes in the genotype of the individual are updated in the final stage of both the compress and expand operators so that all the connections remain intact. The reasons for this is that, the compress and expand operators only make a structural change to the genotype of an individual and have no affect on genotype fitness, as the genotypes before and after the action of these operators represent the same directed graph. The expand operator has twice the probability of being applied to the genotype than the compress operator. We found that this introduces a pressure for good modules to replicate quickly in the genotype of an individual in order to survive. This can be seen as survival-of-the-fittest modules within the genotype itself.

Modules can replicate within the genotype of an individual through the action of the genotype point mutation operator. This is identical to that used in CGP with the exception that it can mutate the function of a node to any of the primitive functions or any available modules in the module list. If a node is mutated to represent a module it is classed as a type II node and is treated like a standard node. This means the genotype point mutation operator can also mutate the function of a type II node to any of the pre-defined functions or any available modules in the module list. It can also mutate any of the inputs of the type II node in the same way it would mutate the inputs of a standard node. If the function of a standard node or type II node is mutated, the new node keeps however many of the original nodes inputs it needs and randomly generates any extra inputs it may require. Type II nodes are also immune from the expand operator as this could cause excessive growth of the genotype that could possibly lead to bloat.

To summarize the properties of node types I and II are shown in Table 1. The main reasons for the two types of module is to try and reduce the excessive growth of the genotype and to also help induce a selection pressure on the modules so that they have to replicate in the genotype (i.e. make the transition from being represented by type I to type II nodes) and be associated with a high fitness genotype in order to sur-

vive. Once the module is represented by a type II node it is harder for the module to be removed from the module list, as it has a lower probability that it will be removed from the genotype (i.e. it cannot be expanded). This is both advantageous as it allows good modules to stay in the module list but is also disadvantageous as it could possibly allow the evolution of the genotype to progress a lot slower.

**Table 1.** Nodes types and their properties.

| Node Type | Action of Compress | Action of Expand | Action of Genotype Point Mutation |
|---|---|---|---|
| I | Creation | Destruction | Change node inputs |
| II | Immune | Immune | Creation or destruction or change node inputs |

The module genotypes contained in the module list can also be evolved through the action of five different operators: *module point mutation, add-input, add-output, remove-input* and *remove-output*. The module point mutation operator is a restricted version of the CGP genotype point mutation operator, as it can still mutate the inputs and function of any node contained in the module genotype but it is not allowed to introduce any type II nodes into the module genotype. It can also mutate which node output each of the module outputs are connected to.

The add-input and add-output operators allow greater connectivity to and from the contents of a module by increasing the number of module inputs or module outputs by one respectively each time either operator is applied, making a more generalized module. When the add-input operator is applied to a module, the gene representing the number of module inputs in the module header part of the module genotype is incremented by one and an extra gene is inserted into all nodes (type I and type II) representing the module in the genotype of the individual, as a randomly chosen value for the new module input. Likewise, when the add-output operator is applied to a module, the gene representing the number of module outputs in the module header part of the module genotype is incremented by one and two extra genes are added to the module output section of the module genotype, as randomly chosen values for the node index and node output that the new module output is connected to.

Alternatively, the remove-input and remove-output operators reduce the connectivity to and from the contents of a module, by decreasing the number of module inputs or module outputs by one respectively each time either operator is applied, therefore making a more specialized module. When the remove-output operator is applied to a module, the gene representing the number of module inputs in the module header part of the module genotype is decremented by one and the gene corresponding to the module input randomly chosen is removed from all nodes (type I and type II) representing the module in the genotype of an individual. Likewise, when the remove-output operator is applied to a module, the gene representing the number of module outputs in the module header part of the module genotype is decremented by one and the two genes corresponding to the randomly chosen module output are removed from the module output section of the module genotype. All of the operators: add-input, add-output, remove-input, and remove-output must comply with the restrictions on the number of module inputs and module outputs at all times. Further information

about all of the module operators (including figures explaining their operation) is available in our previous work [14].

## 4 Product Reduction (PR)

In digital multipliers we require $n^2$ AND gates to compute the product bits. Product reduction (PR) assumes that these have already been provided. It uses the outputs of these gates as inputs to the remaining circuit (which is evolved). PR transforms the standard truth table of $2^{2n}$ rows to an input-output table having $2^{2n} - 2(2^n) - 2$ rows. The width of the PR table is increased from the $2n$ inputs found in the standard truth table to $n^2$ inputs because $n^2$ AND Boolean functions are required to produce the product of every combination of bits. The length of the PR table however is reduced because the PR table contains multiple row entries all containing zeros due to multiplication by 0, which can be reduced to a single row.

## 5 Experiment Details

The performance of CGP and ECGP both with and without PR was tested on the digital multiplier problem (2x2 and 3x3 bit). The fitness is defined as the number of phenotype output bits that differ from the perfect n-bit digital multiplier. A perfect solution has score zero.

The parameter settings used for CGP and ECGP in all of the experiments are shown in Table 2. The probability values chosen for the ECGP operators were found to be optimal by a trial and error process in previous ECGP experiments.

**Table 2.** Parameter settings used for CGP and ECGP in all of the experiments. The operator rate is expressed as a percentage of the genotype length. Both the operator rates and probabilities are per generation. 50 independent runs used.

| Parameter | Value |
|---|---|
| Population size | 5 |
| Initial genotype size | 200 nodes (600 genes) |
| Function set | {AND, AND with one input inverted, OR, XOR} |
| Genotype point mutation rate | 3% (18 Genes) |
| Genotype point mutation probability | 1 |
| Compress/Expand probability | 0.1/0.2 |
| Module point mutation probability | 0.04 |
| Add/Remove input probability | 0.01/0.02 |
| Add/Remove output probability | 0.01/0.02 |
| Maximum module size (ECGP only) | 5 or 10 nodes |
| Module list initial state (ECGP only) | Empty |

## 6 RESULTS

For all experiments, the Computational Effort (CE) was calculated using the formula found in Fig. 6 [3] with z=99% and are shown in Table 3. They are only relevant when comparing CGP and ECGP with the same number of nodes in their genotypes and the same rate for the genotype point mutation operator because CE figures for CGP and ECGP vary significantly depending on these values, therefore potentially causing an unfair comparison. We have only compared the CE figures of ECGP with CGP because no other researchers have provided CE figures for their GP techniques on these problems.

$$P(M,i) = \frac{N_s(i)}{N_{total}}, \quad R(z) = ceil\left\{\frac{\log(1-z)}{\log(1-P(M,i))}\right\}, \quad I(M,i,z) = MR(z)(i+1)$$

**Fig. 6.** The Computational Effort (CE) formula from [3] where $i$ represents the generation number, $N_s(i)$ represents the number of successful runs by generation $i$, $N_{total}$ represents the total number of runs and $M$ represents the number of individuals in the population. $P(M,i)$ represents the cumulative probability of success, $R(z)$ represents the number of independent runs required to give a probability of success $z$ by generation $i$ and $I(M,i,z)$ represents the minimum number of individuals which must be processed to give a probability of success $z$ by generation $i$.

**Table 3.** The CE figures for CGP and ECGP for the digital multiplier problems with and without product reduction. The maximum module size is shown in brackets.

| | **2-Bit Multiplier** | **3-Bit Multiplier** |
|---|---|---|
| **CGP** | 37,600 | 18,509,600 |
| **CGP with PR** | 5,600 | 2,498,800 |
| **ECGP (5)** | 46,000 | 8,400,400 |
| **ECGP with PR (5)** | 6,000 | 1,560,400 |
| **ECGP (10)** | 61,600 | 2,795,200 |
| **ECGP with PR (10)** | 7,600 | 688,800 |
| **CGP-PR Speedup** | 6.7 | 7.4 |
| **ECGP-PR (5) Speedup** | 7.7 | 5.4 |
| **ECGP-PR (10) Speedup** | 8.1 | 4.1 |

For both of the digital multipliers tested over all fifty runs, both CGP and ECGP with and without PR produced 100% successful solutions. The results from both multipliers clearly show that CGP with PR performs between 6.7 and 7.4 times faster than CGP without PR and that ECGP with PR performs between 7.7 and 5.4 (with a maximum module size of five) or 8.1 and 4.1 (with a maximum module size of ten) times faster than ECGP without PR depending on the chosen maximum module size. We note that, rather unexpectedly, the speedup with CGP increases with problem difficulty, while the opposite is true with ECGP where the speedup decreases. We think this is because most of the time taken by CGP without PR to find a solution is used organizing the AND Boolean functions in the 1-bit multiplication section of the circuit. However, ECGP without PR finds the 2x1-bit Multiplier module and re-uses it to quickly find and organize the 1-bit multiplication section. Therefore by eliminating

the 1-bit multiplication section from the search space by using PR, saves CGP more time than ECGP as the problem scales in difficulty.

Comparing the results of CGP and ECGP (both with and without PR) on the individual problems shows that CGP performs quicker than ECGP on the 2-bit multiplier problem. This could be because the exploration of code in the modules hinders the performance of ECGP on small problems, as the results show that by reducing the maximum module size makes the performance of ECGP closer to that of CGP. However, ECGP does perform substantially better than CGP on the harder 3-bit multiplier problem, suggesting that ECGP may perform better on even larger, more complex problems. This speedup could be because ECGP is building and re-using modules containing useful partial solutions out of the primitive functions such as the 1-bit half adder and the 1-bit full adder. The results also show that for harder problems, ECGP performs better with a larger maximum module size (doubling the maximum module size, halved the computational effort for the 3-bit multiplier). This could be because the more nodes a module has the easier it is to find partial solutions. This is an interesting concept and will be investigated further in future work.

All of the experiments were run on a single processor desktop PC with 512MB of memory. The time taken to complete 50 runs of each problem varied between a few minutes to a few hours depending on problem difficulty and whether PR was used. ECGP only took fractionally longer to complete one thousand generations on any problem than CGP showing that the computational time required for the overhead of module acquisition is quite small and the computational time taken for fitness evaluation (both CGP and ECGP) is by far the dominant factor.

## 7 Conclusion

We have presented for the first time the application of PR with CGP and ECGP on the difficult digital multiplier problem. PR is shown to significantly speedup the performance of CGP and ECGP when compared with CGP and ECGP without PR on both multipliers tested. However, CGP was shown to perform better than ECGP on the simpler 2-bit multiplier problem but ECGP performed better on the harder 3-bit multiplier problem indicating that ECGP may perform substantially better than CGP on even larger problems. This is a promising result for ECGP as the results presented in this paper follow a very similar trend to those found in our previous work [14].

It was also found that the maximum module size chosen for ECGP can drastically affect performance and will be investigated further in future investigations. Currently ECGP does not allow modules within modules. However, we do have a working version of ECGP that allows embedded sub-modules but we are currently investigating the problem of bloat within the embedded sub-modules found in the inactive areas of the module genotype. When a solution is found, we intend to allow embedded sub-modules in future work as this could lead to an even greater boost in performance.

## References


[1] Angeline, P. J. Pollack, J. (1993) Evolutionary Module Acquisition, Proceedings of the 2nd Annual Conference on Evolutionary Programming, pp. 154-163, MIT Press, Cambridge.

[2] Dessi, A. Giani, A. Starita, A. (1999) An Analysis of Automatic Subroutine Discovery in Genetic Programming, GECCO 1999: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 996-1001, Morgan-Kaufmann, San Francisco.

[3] Koza, J. R. (1992, 1994) Genetic Programming I and II. MIT Press, London.

[4] Miller, J. F., Thomson, P., and Fogarty T. C. (1997) Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study, Genetic Algorithms and Evolution Strategies in Engineering and Computer Science: Recent Advancements and Industrial Applications. Editors: D. Quagliarella, J. Periaux, C. Poloni and G. Winter, Wiley.

[5] Miller, J. F. (1999) An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach, GECCO 1999: Proceedings of the Genetic and Evolutionary Computation Conference, Orlando, Florida, pp 1135-1142, Morgan Kaufmann, San Francisco.

[6] Miller, J. F. and Thomson, P. (2000) Cartesian Genetic Programming, Proceedings of the 3rd European Conference on Genetic Programming, Edinburgh, Lecture Notes in Computer Science, Vol. 1802, pp 121-132, Springer-Verlag, Berlin.

[7] Miller, J. F., Job D., and Vassilev, V. K (2000) Principles in the Evolutionary Design of Digital Circuits – Part I, Genetic Programming and Evolvable Machines, Vol. 1, pp. 8-35.

[8] Rosca, J. P. (1995) Genetic Programming Exploratory Power and the Discovery of Functions, Proceedings of the 4th Annual Conference of Evolutionary Programming, San Diego, pp 719-736, MIT Press, Cambridge.

[9] Spector, L. (1996) Simultaneous Evolution of Programs and their Control Structures, Advances in Genetic Programming II, pp. 137-154, MIT Press, Cambridge.

[10] Spector, L. (2001) Autoconstructive Evolution: Push, PushGP, and Pushpop, Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001, pp. 137-146. San Francisco, CA: Morgan Kaufmann Publishers

[11] Torresen, J. (2003) Evolving Multiplier Circuits by Training Set and Training Vector Partitioning, Proceedings of the 5th International Conference on Evolvable Hardware, ICES03, Lecture Notes in Computer Science, Vol. 2606, pp. 228-237, Springer-Verlag, Berlin.

[12] Torresen, J. (2004) Exploring Knowledge Schemes for Efficient Evolution of Hardware, Proceedings of the 2004 NASA/DoD Conference on Evolvable Hardware (EH-2004), pp. 209-216, IEEE Comp. Society Press.

[13] Vassilev, V. K. and Miller J. F. (2000) Scalability Problems of Digital Circuit Evolution, Proceedings of the 2nd NASA/DOD Workshop on Evolvable Hardware, pp. 55-64, IEEE Comp. Society Press.

[14] Walker, J. A. Miller, J. F. (2004) Evolution and Acquisition of Modules in Cartesian Genetic Programming, Proceedings of the 7th European Conference on Genetic Programming, Lecture Notes in Computer Science, Vol. 3003, pp 187-197, Springer-Verlag, Berlin.