Advanced Techniques for the Creation and Propagation of Modules in Cartesian Genetic Programming

Paul Kaufmann and Marco Platzner
University of Paderborn
{paul.kaufmann, platzner}@upb.de

ABSTRACT

The choice of an appropriate hardware representation model is key to successful evolution of digital circuits. One of the most popular models is cartesian genetic programming, which encodes an array of logic gates into a chromosome. While several smaller circuits have been successfully evolved on this model, it lacks scalability. A recent approach towards scalable hardware evolution is based on the automated creation of modules from primitive gates.

In this paper, we present two novel approaches for module creation, an age-based and a cone-based technique. Further, we detail a cone-based crossover operator for use with cartesian genetic programming. We evaluate the different techniques and compare them with related work. The results show that age-based module creation is highly effective, while cone-based approaches are only beneficial for regularly structured, multiple output functions such as multipliers.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*Program Synthesis*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Cartesian genetic programming (CGP), embedded cartesian genetic programming (ECGP), automatically defined functions (ADFs), module acquisition, crossover operator

1. INTRODUCTION

Evolvable hardware [5, 9] combines evolutionary algorithms with reconfigurable hardware in order to construct smaller, more robust, or even self-adaptive and self-optimizing hardware systems. The common denominator of all evolvable hardware approaches is the application of evolutionary techniques directly at the hardware level. Here, hardware means both digital and analog electronic circuits, and the hardware level comprises all models of hardware, from configuration bitstreams for reprogrammable devices over netlists of gates to behavioral descriptions. Evolutionary techniques have been shown to be able to generate astonishing circuits that are totally different from classically engineered circuits, and sometimes even superior [24]. Moreover, for applications with time-varying specifications very promising initial results have been achieved that indicate the potential of evolutionary techniques to construct self-adapting systems. Examples include evolved controllers for prosthetic hands and robot navigation [8].

The choice of a suitable hardware representation model, i.e., the encoding of a hardware circuit into a chromosome, is key to a successful application of evolutionary design techniques. On one hand, the representation determines the size and the structure of the search space which affects the efficiency of the evolutionary operators. Using a low-level representation, e.g., logic gates and wires, the chromosome length grows rapidly with the circuit size. This might render evolutionary techniques infeasible and is known as the *scalability problem*. On the other hand, high-level representations, e.g., behavioral descriptions, require substantially more effort to map an individual to the target hardware. The majority of related work in evolvable hardware focused on creating combinational Boolean functions. Typically, the chromosome encodes a netlist with a number of logic nodes and an interconnect. A rather popular representation model arranges the logic gates in an array which resembles the architecture of current programmable hardware devices, such as field-programmable gate arrays (FPGAs). As most authors implemented versions of genetic programming on this model, the approach has been termed *cartesian genetic programming* (CGP).

For larger and real-world applications, the CGP model in combination with fine-grained logic lacks scalability. The excessive chromosome lengths lead to extremely large search spaces. To tackle the problem of scalability, a number of approaches have been proposed which can be classified along three dimensions: The first dimension corresponds to the level of the hardware representation. To improve scalability we should give up too hardware-friendly structural models and move towards behavioral models. Orthogonally to that, we see the dimension of object granularity. We can evolve hardware using gates and wires, arithmetic functions and buses, and eventually complete intellectual property cores and networks on chip. The third dimension relates to incorporating knowledge into the evolutionary process. We can

GECCO'08, July 12–16, 2008, Atlanta, Georgia, USA.

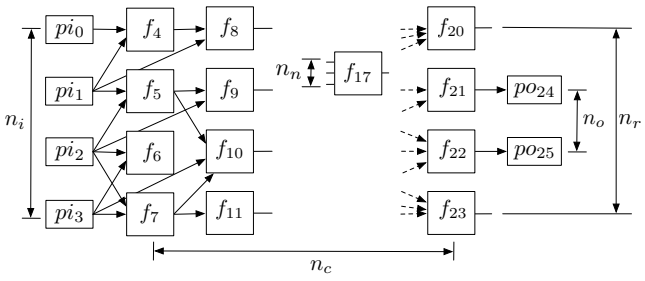

Figure 1: Cartesian genetic programming (CGP) model and its parameters

either evolve in a blind manner or try to leverage domain or problem-specific knowledge. An example for using problem-specific knowledge is demonstrated in [26]. There, the fact is exploited that multiplier circuits usually compute products in the first few logic levels and add partial sums in later levels. Consequently, the representation model was constrained such that `and` gates were preferred in the first columns of the logic array and `adders` in the remaining columns.

The main contribution of this paper is the introduction of novel techniques for identifying and dealing with modules, leveraging a previously discussed approach for module creation in the CGP model [28]. We present and evaluate two module creation techniques, called *age-based* and *cone-based* module creation. Further, we develop and evaluate a novel *cone-based crossover* operator to be used with a genetic algorithm on the CGP model. In a sense, we address all three dimensions discussed above. First, we are giving up the structural, hardware-friendly aspects of the CGP model. Second, modules are building blocks of coarser granularity, albeit generated in an automated fashion. Finally, our cone-based approach represents circuit design knowledge.

The paper is structured as follows: In Section 2 we introduce the cartesian genetic programming model as well as its extension to automated module creation and review related work on these models. Our novel methods for improving the automated module creation and propagation process are presented in Section 3. Section 4 describes the experimental setup to evaluate the proposed techniques and presents and discusses our empirical findings. Finally, we summarize and point to future work in Section 5.

2. RELATED WORK

In this section we review related work on genetic programming models for digital circuit evolution. In particular, we discuss cartesian genetic programs for representing digital circuits and their extension to automated module creation. Further, we point to the fundamental trade-off between an efficient evolution and the effort required to map evolved circuits to real hardware.

2.1 Cartesian Genetic Programs

A cartesian genetic program (CGP) is a structural hardware model that arranges logic cells in a two-dimensional geometric layout [19, 18]. In contrast to a genetic program [13, 14] which relies on trees to represent evolved functions, a CGP essentially is a restricted directed acyclic graph (DAG). The restrictions are posed by the array structure which limits the number of overall logic cells and the depth of an evolved circuit.

Formally, a CGP model consists of $n_c \times n_r$ combinational logic blocks, n_i primary inputs, and n_o primary outputs. A logic block has n_n inputs and implements one out of n_f different logic functions of these inputs. While the primary inputs and outputs can connect to any logic block input and output, respectively, the connectivity of the logic block inputs is restricted. The input of a logic block at column c may only connect to the outputs of blocks in columns $c-l, \ldots, c-1$ as well as to the primary inputs. The levels-back parameter l restricts wiring to hardware-friendly local connections. More importantly, as only feed-forward connections are allowed the creation of combinational feedback loops is avoided. Figure 1 shows an example for a CGP model together with its parameters. The model in this example has five columns, four rows, four primary inputs, and two primary outputs.

An individual circuit is defined by its chromosome (genotype). The length of a chromosome is constant and given by $n_c \cdot n_r(n_n + 1) + n_o$ integer values. Each logic block in the array is characterized by $n_n + 1$ values, one for each input and one for the logic function. Additionally, an n_o-tuple of values selects the block outputs that are connected to the primary outputs of the circuit. The logic block genes are mapped to the array in order of their position within the chromosome. Consequently, a CGP implicitly encodes the placement of logic blocks.

CGP models have been used with different sets of node functions to evolve a variety of digital circuits. Simple logic gates have been used to evolve logic and arithmetic circuits such as adders, parity functions, comparators and multipliers [27, 17], hashing functions [4], as well as controllers for prosthetic hand control [10, 25] or robot navigation [23, 11, 15]. Higher-level functional elements such as adders and multipliers have been applied to evolve more complex signal processing applications, for example in data compression [20] or cell scheduling [16].

Further, CGP models have been used with different evolutionary strategies, from classic genetic algorithms, e.g., [4], to evolutionary strategies, e.g., [28, 29]. An often-used mutation operator on the CGP model is *one-point mutation*. One-point mutation randomly selects a node and modifies either the node function or one node input. If the function is mutated, a new function is chosen from the function set randomly. If an input is modified, one of the node's inputs is ripped up and reconnected to the output of a randomly chosen node in one of the previous columns or to a primary input. The straight-forward implementation of a crossover operator acts on the geometrical structure of the chromosome. In the case of *uniform crossover*, a new individual is created by selecting its nodes from the corresponding geometrical node positions of two parents with equal probability. An *n-point crossover* divides the parents' chromosomes into $n + 1$ parts based on the numbering of the nodes. The child's chromosome is then constructed by alternatively selecting partial chromosomes from the parents.

In CGP, nodes which do not contribute to the primary outputs remain in the chromosome and might be propagated through the generations. This property, termed *neutrality*, has been shown to improve the convergence of the search process [18]. The main problem with the CGP model is scalability. In the last years, a number of approaches have been presented that address this problem. One of them is

function-level evolution which uses node functions of coarser granularity and buses instead of single bit wires, e.g., in [22, 20, 16]. Another approach seeks to automatically evolve such coarse-grained blocks (modules) from primitive functions. While techniques for the automated creation of modules have been applied in genetic programs for some time, their use in CGP models has been demonstrated only recently [28]. This technique is described in Section 2.3 in more detail.

2.2 FPGA Mapping

An intriguing feature of the original CGP model is its closeness to FPGA hardware. As CGP uses logic cells with one output and implicitly encodes placement, the actual process of mapping an evolved chromosome to a lookup-table based FPGA reduces to routing and bitstream generation. While most related work in CGP aims at evolutionary circuit design where hardware mapping is required only once after an appropriate circuit has been evolved, there are two cases for which an efficient FPGA mapping is of utmost importance. The first is the use of FPGAs as accelerators to speed up the fitness evaluation, e.g., in [6]. The second and presumably more challenging one is on-chip evolution [7, 12]. However, there is a lack of open FPGA bitstream architectures which makes the creation of custom routing and bitstream generation tools a research challenge on its own. For that reason, several researchers resorted to *virtual FPGAs* and implemented routing by controlling multiplexers that select between different datapaths [21, 6].

Generally, there exists a trade-off between the chromosome length and thus the efficiency of evolution, and the effort needed for mapping a chromosome to an FPGA. Given modern FPGA architectures and their corresponding design tools, it can be doubted whether the implicit encoding of placement as done in CGP is actually useful. As discussed in [1], CGP encodes information that influences the characteristics of the phenotype (FPGA circuit), but does not effect the fitness (combinational behavior). This could be seen as a source of inefficiency. This position is underlined by recent work that resorts to a one-row CGP model, e.g., [30]. In the one-row CGP model node inputs are allowed to connect to any previous node and the primary inputs, i.e., $l = n_c$. In comparison to an array of nodes, the one-row approach features higher compactness. As an example, consider the evolution of a one output function with three logic blocks connected in a series. To be able to evolve such a function with the original array model, we have to provide at least three columns of nodes, whereas in the one-row model we can afford an overall smaller number of nodes. The reduced chromosome length of the one-row model makes the evolutionary operators more efficient. While the one-row model gives up the implicit encoding of placement information in the chromosome and thus makes FPGA mapping more involved, it still shows important properties of the original array-based CGP model, including a bounded number of nodes and bounded depth, as well as neutrality.

Recently, a crossover operator for the one-row CGP model was shown in [2]. This crossover operator requires to map the CGP chromosome to a string of real values. To that end, the n_c node numbers are mapped to n_c intervals in $[0, 1]$, and the n_f functions are mapped accordingly. The operator acts on two strings of real values between $[0, 1]$ and creates a child by a linear combination of the parents' values. The authors observed an improved initial convergence rate of the optimization process and presented a scheme with an adaptive crossover rate.

2.3 Automated Module Creation

In [28], a technique for automated module creation in the one-row CGP model was introduced. Modules are compositions of primitive node functions which are Boolean gates. The resulting hardware representation model has been termed *embedded CGP (ECGP)*. The number of primitive nodes of a module is restricted by lower and upper bounds. A module must have at least two inputs and one output. ECGP models are non-hierarchical in the sense that modules can not contain other modules.

The automated creation and use of modules is governed by three operators: *compress, expand* and *module mutation*. Compression replaces a number of primitive nodes by a newly created module. All inputs to module nodes that originate from non-module nodes or primary inputs become module inputs, and all outputs of nodes in the module that target non-module nodes or primary outputs become module outputs. The function and routing of nodes within the module remain unchanged. Hence, applying the compress operator to a chromosome does not change its fitness. Importantly, in [28] nodes with subsequent node numbers in the one-row CGP representation are selected to form a new module. As a node can connect to any previous node and the primary inputs ($l = n_c$), the compress operator basically selects nodes randomly. Expand is the reverse process of compress and replaces a module with its primitive nodes, again leaving the node functions and the routing unchanged.

A module in the evolved circuit points to a corresponding module description. The module descriptions are treated as separate and self-contained sub-genotypes which are stored in a list at the end of the ECGP chromosome. Multiple modules in the circuit may refer to the same module description, which effectively enlarges the set of available node functions. Modules created by compress are denoted as type I modules. Additionally, modules can also be created by the one-point mutation operator applied to primitive nodes. One-point mutation cannot create a new module description but it can replace a primitive node function with a reference to one of the already created modules in the modules list. Such modules are denoted as type II modules. Only type I modules can be expanded and only type II modules can be mutated by the one-point mutation operator. The module descriptions themselves can be modified by the module mutation operator which is essentially a standard CGP mutation operator, except that it operates on module descriptions rather than on the overall chromosome and can additionaly add and remove modules inputs and outputs.

3. ADVANCED TECHNIQUES

In our work we leverage ECGP, the hardware representation model introduced in [28]. A circuit is described as a DAG with a restricted number of nodes. The chromosome encoding relies on the one-row CGP approach where all nodes are brought into a linear order and connections, while limited to feed-forward wires, can span the entire row. Figure 2 shows an example for such a circuit. Some of the nodes are primitive node functions (nodes f_5, f_6, f_8, f_9, and f_{10}), and some are modules (nodes m_7 and m_{11}) which refer to a corresponding module description. While we implement

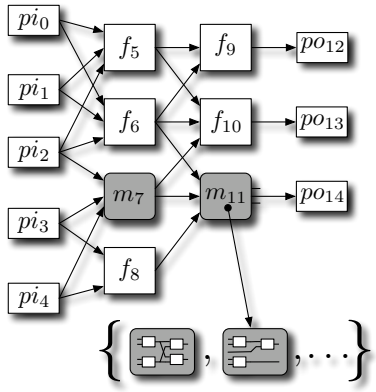

Figure 2: ECGP hardware representation model

the one-row CGP model to encode chromosomes, the graphical circuit representation of the DAGs we use in this paper is column-oriented. Each column represents a new level of logic, with the exception of modules that can contain several logic levels themselves. The order of nodes, primitive ones and modules, in the linearized one-row CGP model is uniquely determined by the node numbers. After applying compress and expand operators we adapt the node numbering to ensure that the following property always holds: Given any two nodes (primitive nodes and modules) h_i and h_j with $i < j$, an input of h_i can never connect to h_j.

In this section, we introduce three new techniques for identifying and dealing with modules. The first two techniques are *age-based* and *cone-based* module creation and try to improve the previous module creation technique which basically selects primitive nodes randomly. Then, we investigate a *cone-based crossover* operator which allows us to experiment with a genetic algorithm instead of previous work's $1 + \lambda$ evolutionary strategy.

3.1 Age-based Module Creation

Age-based module creation aggregates primitives nodes that have persisted in the chromosome for a higher number of generations. The rationale behind age-based module creation is that aged nodes are likely to contribute directly or indirectly to an individual's success and should therefore be preferred over randomly selected nodes.

We assign to each primitive node f_i an attribute $age(f_i)$. The age is incremented by one in each generation and set to zero when the node is selected for mutation or compression. The age of primitive nodes within modules remains unchanged; modules themselves do not have an age. We form module candidates by aggregating primitive nodes, restricting the number of nodes by lower (n_{min}) and upper bounds (n_{max}). The average age of a module candidate m_j is then given by

$$\text{age}(m_j) = \frac{\sum_{f_i \in m_j} age(f_i)}{|m_j|}$$

In our current implementation of the age-based module creation technique we use two-stage binary tournament to select a module that is actually created. That is, we generate a module candidate by following procedure: First, we select a random primitive node f_i and a number of primitive nodes $n, n_{min} \leq n \leq n_{max}$, randomly. Then, we extend the module from f_i to nodes with smaller node numbers until we hit a module or aggregate exactly n primitive nodes. We create another module using a different random primitive node f_i and draw the one with higher average age. If both modules have the same average age, we draw one module randomly. This step is repeated once to derive the final module.

We have also experimented with selecting the module candidate with maximum average age. This requires the formation and evaluation of a larger number of module candidates. For example, consider the circuit shown in Figure 2 with its five primitive nodes and two modules. As the encoding uses the one-row CGP model, there are the following three modules of size two: $(f_{10}, f_9), (f_9, f_8), (f_6, f_5)$, and one module of size three: (f_{10}, f_9, f_8). However, picking the module with maximum average age has proven inferior to the two-stage binary tournament scheme for all test problems. An explanation for this lies in the fact that maximizing average module age tends to generate modules with a very small number of high-aged nodes. It seems that while using node age as a guide to steer module creation is highly effective (see Section 4), the technique is rather sensitive to the size of modules. A deeper investigation of this dependency remains to be done in further work.

3.2 Cone-based Module Creation

Both age-based module creation as well as the original module creation described in [28] aggregate primitive nodes into modules without taking the connections between nodes into consideration. In contrast, cone-based module creation aggregates only primitive nodes that are within a circuit structure called *cone*. Cones are a widely-used concept in circuit synthesis, especially in the area of lookup-table mapping for FPGAs (see, for example, [3]). Given a node f_r in the DAG, a cone rooted at f_r consists of f_r itself plus some predecessor nodes such that for any node f_i in the cone there exists a path from f_i to f_r that is entirely in the cone. For example, in Figure 3(a) the node set (f_{11}, f_9, f_8) forms a cone. Note that while a cone has a distinct root node f_r, it can have several outputs. The rationale behind cone-based module creation is that many useful substructures in classically engineered circuits are cones, e.g., the sum and carry functions of a full adder.

To generate module candidates, we randomly select a primitive node f_i and create a cone rooted at f_i with a number of nodes randomly chosen between n_{min} and n_{max}.

One subtlety in generating cones is that we have to avoid what is called *reconvergent paths* in logic synthesis. To discuss several subtypes of cones consider again the circuit in Figure 3(a). The node set (f_{11}, f_9, f_6) is called a fan-out free cone because the fanout (output connections) of every node except the root node stay within the cone. Such cones are certainly safe candidates for module creation. The node set (f_{11}, f_9, f_8, f_5) does not form a fan-out free cone, as the outputs of f_5 and f_8 leave the cone. Nevertheless, this node set forms a valid module. In contrast, the node set (f_{10}, f_8, f_5) which is highlighted in Figure 3(a) does not form a valid module as the output of f_5 leaves and reenters the cone. If this cone was turned into a module the resulting circuit, shown in Figure 3(b), would contain a combinational feedback loop. The path formed by nodes (f_5, f_7, f_{10}) is called reconvergent with respect to the cone (f_{10}, f_8, f_5).

Reconvergent paths are specific to the cone-based module

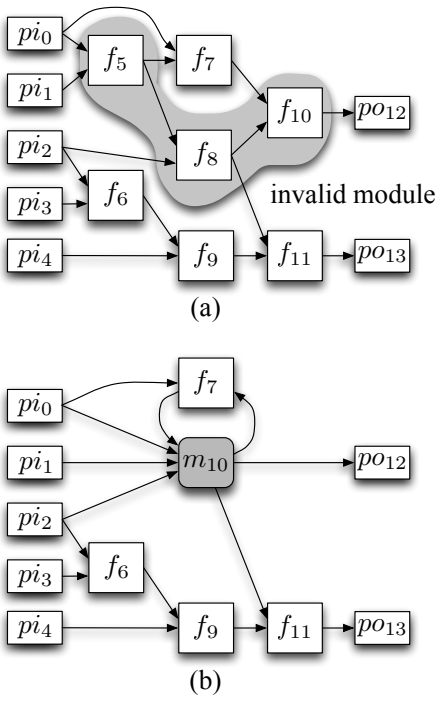

Figure 3: Cones with reconvergent paths are invalid (a) as they can lead to combinational feedback loops (b)

creation technique. Neither the age-based technique nor the original method of [28] can create such paths. Compared to module creation methods in [28], which relies on contiguous node numbers, we are using breadth first search starting with the cone's root node to avoid reconvergent paths. Resuming the example of Figure 3(a), a cone of size 3, rooted at node f_{10} would be formed by the nodes (f_{10}, f_7, f_8).

3.3 Cone-based Crossover

In order to compare genetic algorithms to the previously applied $1 + \lambda$ evolutionary strategies we have developed a cone-based crossover operator. Our genetic algorithm uses the ECGP hardware representation model and the same operators as described in [28], including compress and expand. Additionally, we employ a crossover technique that generates a recombined chromosome by transplanting a cone of a donor chromosome into a clone of a recipient chromosome.

In the first step, we form a cone in the donor chromosome by randomly selecting a root node and a size between n_{min} and n_{max}. This procedure is similar to the cone-based module creation of Section 3.2 except that we treat both primitive nodes and modules as atomic nodes. Note that at this point, a cone can contain modules. In the second step, we randomly select a root node in the recipient and try to form a cone of exactly the same size as the donor's cone. Depending on the actual DAGs, the resulting cone of the recipient can be smaller than the donor's cone. The third step comprises the formation of two sets, set p that contains nodes of the donor's cone which have output connections to nodes outside the cone, and another set q that contains the nodes of the recipient which connect to nodes within the recipient's cone. Figure 4 displays an example. The donor's cone consists of three nodes. As all these nodes provide cone outputs, we derive $p = \{f_{10}, f_{11}, f_{22}\}$. The recipient's cone receives inputs from three nodes, and we derive $q = \{f_5, f_7, f_8\}$. The fourth step actually transplants the donor's cone into a clone of the recipient, forming a new recombined chromosome. This process preserves all node types. Specifically, nodes in the donor's cone which are modules of type I or II remain modules of type I or II, respectively. The module descriptions of the recombined chromosome are updated accordingly. In the final step, dangling inputs of the transplanted module and the recepient chromosome are randomly connected to the nodes in lists p and q, respectively. If the resulting chromosome still contains unconnected inputs, they are connected randomly to predecessing nodes.

4. EXPERIMENTS AND RESULTS

In this section, we present the experimental setup including the parameters of the ES and GAs, the test problems, and the metrics that is reported. Then, we show the results and discuss the findings from our experiments.

4.1 Evaluated Metrics

As metrics to compare the proposed techniques we use the *computational effort* as presented by Koza in [14]. For each experiment with its specific number of M fitness evaluations per generation, a number of independent runs is conducted. In each run the optimization goal, i.e., the evolution of a functionally correct circuit, will be reached by some generation i. The probability of reaching the optimization goal by generation i can then be expressed as follows:

$$P(M, i) = (\#\text{succeeded runs by generation } i)/(\#\text{runs})$$

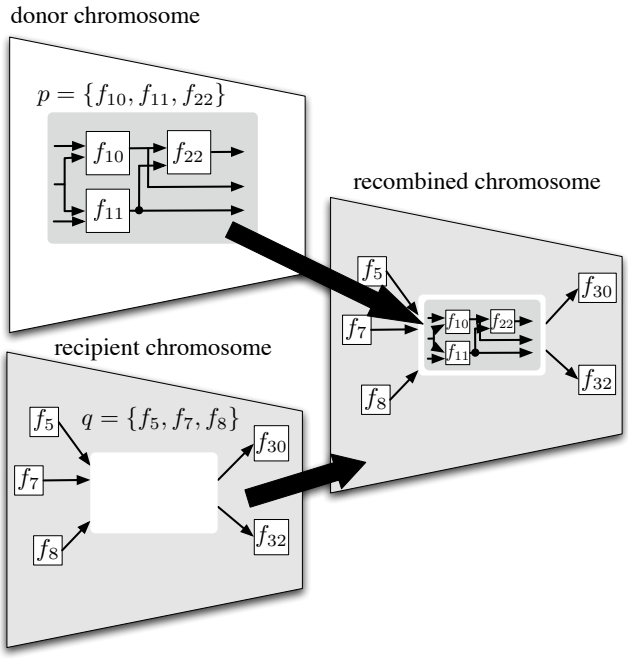

Figure 4: Cone-based crossover: A cone of a donor chromosome is transplanted into a clone of a recipient chromosome.

	test problems		
	parity	multiplier	emg classifier
chromosome length	50 nodes	200 nodes	200 nodes
number of inputs n_i	3/4/5	4/6	200
number of outputs n_o	1	4/6	1
functional set	2-LUT: and, nand, or, nor	4-LUT: and, and$_{inv}$, or, xor	4-LUT: any function
M for $1+4$ ES / GA-5/ GA-50	4/4/47	4/4/47	4/4/47
mutation rate	0.03	0.03	0.03
one-point mutation probability	0.6	0.6	0.6
compress/expand probability	0.1/0.2	0.1/0.2	0.1/0.2
module mutation probability	0.1	0.1	0.1
module size	2...8 nodes	2...10 nodes	2...10 nodes
recombination probability for GA	0.01	0.01	0.01
crossover cone size	2...20 nodes	2...20 nodes	2...20 nodes

Table 1: ECGP parameters for the different test problems

From that we can determine $R(z)$, the number of independent runs that have to be conducted to reach the optimization goal with a certain probability z:

$$R(z) = \lceil \log(1-z)/\log(1-P(M,i)) \rceil$$

The estimated overall number of fitness evaluations required to reach the goal with probability z is then set to:

$$I(M,I,z) = M \cdot (i+1) \cdot R(z)$$

For each experiment with given M and z, the minimal value for $I(M,i,z)$ is determined as the computational effort of the experiment. In our experiments, we have set z to 99% and repeated all experiments for 50 times.

4.2 Test Problems

To be able to compare our techniques with the approaches presented in previous work, we have used even-parity and multiplier functions as test problems and set the ECGP model parameters as given in [28, 29]. To allow for a meanigful evaluation that isolates the effects of our proposed module creation and propagation techniques, we have chosen to implement our own version of the basic ECGP model as a reference, rather than directly comparing our results to that published in [28, 29].

Additionally, we have included classifiers for electromyographic signals as test problems. In this application, skin-attached sensors collect electric signals of contracting muscles to control a prosthetic hand [10, 25]. The test data has been recorded from four muscles of a volunteer's forearm. A sequence of eight contractions (movements) with 20 repetitions each has been measured. The typical signal for a movement is composed of a 9 seconds relax phase and a 5 seconds contraction phase. From the last two seconds of the contraction phase we have removed the dc offset and applied rms smoothing to achieve the feature vectors. The resulting data set consists of 144 strings of 200 bits each. Based on that data we have tried to evolve a classifier circuit for the movement "open hand". Classifiers differ from arithmetic circuits in that there is no simple correctness measure. Typically, classifiers are evolved with training data and then run on test data to determine metrics such as classification rate. As we want to investigate and compare the computational effort for evolving a classifier and not the generalization capabilities of the ECGP model, we have measured the classifiers' fitness on the training data set and defined it to be correct when the classification rate on training data exceeds 85% and 95%, respectively.

The ECGP model parameters, including the chromosome length, the number of inputs and outputs, and the function set for the nodes are shown in Table 1. For the parity function, we have used 2-input lookup table (LUT) nodes but restricted the function set to a few Boolean functions. For the multipliers, we have used 4-input LUTs but again restricted the function set to the functions and, or, xor, as well as and$_{inv}$, which is an and with one input inverted. Finally, for the emg classifiers we have used 4-LUTs without any restriction on the node function.

4.3 ES and GA Setup

We have implemented a 1+4 ES, where the fittest individual of a generation proceeds to the next generation. In case a parent and an offspring have equal fitness, the offspring is promoted over the parent. Further, four clones of this individual are created and mutated. Mutation is applied with a probability of one, but splits into three cases. We either apply one-point mutation, compress/expand followed by one-point mutation, or module mutation. The expansion operator is more likely executed than the compress operator to increase the pressure towards useful modules. The mutation rate denotes the percentage of mutated nodes in a circuit or module, respectively. The actually used rates and probabilities can be found in Table 1.

We have also implemented a standard elitism-based GA with binary tournament selection. The elitism rate is 5%, but at least one individual is picked. In the GA with population size of five (GA-5), the best individual proceeds directly to the next generation. For a population size of 50 (GA-50), the three best individuals proceed directly to the next generation which leaves us with 47 remaining fitness evaluations. The cone-based crossover operator considers cones with a size of up to 20 nodes (primitive nodes and modules).

4.4 Discussion of Results

The experimental results are presented in Tables 2 and 3. The comparison of the different module creation techniques is shown in Table 2, and the comparison between the ES and GAs is shown in Tables 3. Both tables report the computational effort in absolute numbers and relative to our reference implementation which uses the techniques presented previously [28]. A negative relative effort denotes an improvement. For the comparison between ES and GA, we do not provide results for the 5-parity function due to long simulation times. From the experimental results, we can make the following observations:

	computational effort				
	analogue to previous work [28]	aging		cone	
	absolute	absolute	relative to [28]	absolute	relative to [28]
2x2 mul	66,623	51,961	**−22.0**%	49,052	**−26.4**%
3x3 mul	8,840,574	6,001,917	**−32.1**%	3,638,120	**−58.9**%
3-parity	81,122	49,160	**−39.4**%	87,915	+8.4%
4-parity	477,880	494,295	+3.4%	265,796	**−44.4**%
5-parity	1,825,645	1,385,244	**−24.1**%	1,112,691	**−39.1**%
85% emg classifier	18,260	14,743	**−19,3**%	23,855	+30.7%
95% emg classifier	510,147	314,311	**−38,4**%	873,319	+71.2%

Table 2: Experimental results: $1+4$ ES with different module creation techniques

	computational effort				
	$1+4$ ES, analogue to previous work [28]	GA, |population|=5		GA, |population|=50	
	absolute	absolute	relative to [28]	absolute	relative to [28]
2x2 mul	66,623	64,111	**−3.8**%	102,593	+54.0%
3x3 mul	8,840,574	2,518,964	**−71.5**%	39,064,742	+341.9%
3-parity	81,122	382,036	+470.9%	186,898	+130.4%
4-parity	477,880	6,294,678	+1217.2%	6,482,504	+1256.5%
85% emg classifier	18,260	19,859	+8,8%	28,825	+57,9%
95% emg classifier	510,147	576,988	+13.1%	695,794	+36.4%

Table 3: Experimental results: $1+4$ ES versus GA with different population sizes

1. Age-based module creation is highly effective. For six out of the seven test problems, age-based module creation lowers the computational effort in comparison to the previous method with improvements ranging between some 20% and 40%. The one exception is the 4-parity function, where the computational effort increased slightly by 3.4%.

2. The overall results for the cone-based module creation technique are somewhat inconclusive. However, looking at the different test problems we note that for the evolution of multipliers and for larger parity functions cone-based module creation proves highly beneficial. In contrast, for evolving emg classifiers the cone-based approach does not work at all. Intuitively, the identification of cones as useful subcircuits is hampered if the function is rather small or is a single-output function. In the first case there is no sufficient potential for creating cones, whereas the second case lacks reusability of a cone for different outputs. Multipliers are highly regularly structured functions that are neither particularly small nor single-output functions. From the experimental data it is clear that cone-based module creation is effective for multipliers, especially more effective than age-based module creation. In contrast, emg classifier circuits are random logic functions which might explain the unsatisfying performance of cone-based module creation for this class of problems.

3. Comparing the $1+4$ ES to a GA with population size of 5, we conclude that the GA is better for multipliers and dramatically worse for the parity function and for the emg classifiers. Again, this points to the effectiveness of the cone-based approach for multipliers and to its inefficiency for single-output and random logic circuits. Increasing the population size for the GA to 50 increases the computational effort in any case substantially. It has to be noted that a GA with a population size of 50 also evolves correct circuits but needs far more fitness evaluations. In each generation, this GA performs 11.75× more fitness evaluations that the ES and the GA with a population size of 5. As the results show, even for multipliers this larger potential for recombination does not outweigh the higher effort per generation.

5. SUMMARY AND FUTURE WORK

In this paper, we have presented advanced techniques for creating and propagating modules in the CGP hardware representation model. Age-based module creation prefers the aggregation of rather old primitive nodes into modules, cone-based module creation forms modules out of primitive nodes that form cones. Further, we have detailed a crossover operator that selects a cone consisting of both primitive nodes as well as modules in a donor chromosome and transplants this cone into a clone of a recipient chromosome. We have evaluated our novel techniques and compared them to the module creation approach presented in previous work. The results demonstrate the effectiveness of age-based module creation. Cone-based module is even more effective but only for regularly structured multiple output circuits such as multipliers. For smaller and random logic circuits, the cone-based technique should be avoided. Comparing evolutionary strategies with genetic algorithms, we have seen that the cone-based crossover operator is again only beneficial for multipliers. Increasing the size of the population showed an adverse effect on the computational effort.

There are a number of lines for future work. For example, the sensitivity of age-based module creation on the module size should be investigated. Perhaps one can find an optimal range for the size of modules, or adapt the module size to the current module population. Another option for the GA is to let the individuals of a population share the evolved modules, which could improve the convergence behavior. Further, we would like to experiment with the injection of classically engineered modules, such as full adders for the evolution of multipliers. This would naturally lead to mixed-granularity hardware representation models which also fit modern FPGA architectures.

6. ACKNOWLEDGEMENT

This work was supported by the German Research Foundation under project number PL 471/1-2 within the priority program *Organic Computing*.

7. REFERENCES

[1] X. Cai, S. L. Smith, and A. M. Tyrrell. Positional Independence and Recombination in Cartesian Genetic Programming. In *Proceedings 9th European Conference on Genetic Programming (EuroGP)*, volume 3905 of *LNCS*, pages 351–360. Springer, 2006.

[2] J. Clegg, J. A. Walker, and J. F. Miller. A new Crossover Technique for Cartesian Genetic Programming. In *Proceedings of the 9th annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 158–1587. ACM, 2007.

[3] J. Cong and Y. Ding. Combinational Logic Synthesis for LUT Based Field Programmable Gate Arrays. *ACM Transactions in Design Automation of Electronic Systems*, 1(2):145–204, 1996.

[4] E. Damiani, V. Liberali, and A. G. B. Tettamanzi. Evolutionary Design of Hashing Function Circuits Using an FPGA. In *Proceedings International Conference on Evolvable Systems (ICES)*, pages 36–46. Springer, 1998.

[5] H. de Garis. Evolvable Hardware – Genetic Programming of a Darwin Machine. In *Proceedings International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA)*. Springer, 1993.

[6] K. Glette and J. Torresen. A Flexible On-Chip Evolution System Implemented on a Xilinx Virtex-II Pro Device. In *Proceedings 6th International Conference on Evolvable Systems (ICES)*, volume 3637 of *LNCS*, pages 66–75. Springer, 2005.

[7] K. Glette, J. Torresen, and M. Yasunaga. Online Evolution for a High-Speed Image Recognition System Implemented On a Virtex-II Pro FPGA. In *Proceedings of the Second NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 463–470. IEEE Computer Society, 2007.

[8] T. Higuchi and N. Kajihara. Evolvable Hardware Chips for Industrial Applications. *Communications of the ACM*, 42(4):60–66, April 1999. ACM Press.

[9] 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 *Proceedings 2nd International Conference on Simulation of Adaptive Behavior (SAB)*, pages 417–424. MIT Press, 1993.

[10] I. Kajitani, I. Sekita, N. Otsu, and T. Higuchi. Improvements to the Action Decision Rate for a Multi-Function Prosthetic Hand. In *Proceedings 1st International Symposium on Measurement, Analysis and Modeling of Human Functions*, 2001.

[11] D. Keymeulen, M. Durantez, K. Konaka, Y. Kuniyoshi, and T. Higuchi. An Evolutionary Robot Navigation System Using a Gate-Level Evolvable Hardware. In *Proceedings International Conference on Evolvable Systems (ICES)*, pages 195–209, 1996.

[12] D. Keymeulen, M. Iwata, Y. Kuniyoshi, and T. Higuchi. Online evolution for a self-adapting robotic navigation system using evolvable hardware. *Artif. Life*, 4(4):359–393, 1998.

[13] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1993.

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

[15] R. A. Krohling, Y. Zhou, and A. M. Tyrrell. Evolving FPGA-based Robot Controllers using an Evolutionary Algorithm. In *Proceedings of the 1st International Conference on Artificial Immune Systems (ICARIS)*, pages 41–46, September 2002.

[16] W. Liu, M. Murakawa, and T. Higuchi. ATM Cell Scheduling by Function Level Evolvable Hardware. In *Proceedings International Conference on Evolvable Systems (ICES)*, pages 180–192, 1996.

[17] J. Miller. An Empirical Study of the Efficiency of Learning Boolean Functions using a Cartesian Genetic Programming Approach. In *Proceedings Genetic and Evolutionary Computation Conference (GECCO)*, pages 1135–1142, 1999.

[18] J. Miller and P. Thomson. Cartesian Genetic Programming. In *Proceedings 3rd European Conference on Genetic Programming (EuroGP)*, pages 121–132. Springer, 2000.

[19] J. F. Miller, P. Thomson, and T. Fogarty. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In *"Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pages 105–131. John Wiley and Sons, 1998.

[20] M. Salami, M. Murakawa, and T. Higuchi. Data Compression Based on Evolvable Hardware. In *Proceedings International Conference on Evolvable Systems (ICES)*, pages 169–179, 1996.

[21] L. Sekanina. Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware. In *Proceedings 5th International Conference on Evolvable Systems (ICES)*, pages 186–197. Springer, 2003.

[22] L. Sekanina and V. Drabek. Automatic Design of Image Operators Using Evolvable Hardware. In *Proceedings 5th IEEE Design and Diagnostics of Electronic Circuits and Systems*, pages 132–139, 2002.

[23] A. Thompson. Evolving Electronic Robot Controllers that Exploit Hardware Resources. In *Advances in Artificial Life: Proceedings 3rd European Conference on Artificial Life (ECAL)*, volume 929 of *LNAI*, pages 640–656. Springer-Verlag, 1995.

[24] A. Thompson and P. Layzell. Analysis of Unconventional Evolved Electronics. *Communications of the ACM*, 42(4):71–79, 1999.

[25] J. Torresen. Two-Step Incremental Evolution of a Prosthetic Hand Controller Based on Digital Logic Gates. In *Proceedings 4th International Conference on Evolvable Hardware (ICES)*, volume 2210 of *Lecture Notes in Computer Science*. Springer, 2001.

[26] J. Torresen. Evolving Multiplier Circuits by Training Set and Training Vector Partitioning. In *Proceedings 6th International Conference on Evolvable Hardware (ICES*, pages 228–237. Springer, 2003.

[27] V. K. Vassilev, J. F. Miller, and T. C. Fogarty. On the Nature of Two-Bit Multiplier Landscapes. In *Proceedings 1st NASA/DoD Workshop on Evolvable Hardware*, 1999.

[28] J. A. Walker and J. F. Miller. Evolution and Acquisition of Modules in Cartesian Genetic Programming. In *Proceedings 7th European Conference on Genetic Programming (EuroGP)*, volume 3003 of *LNCS*, pages 187–197. Springer, April 2004.

[29] J. A. Walker and J. F. Miller. Improving the Evolvability of Digital Multipliers Using Embedded Cartesian Genetic Programming and Product Reduction. In *Proceedings 6th International Conference on Evolvable Systems(ICES)*, volume 3637 of *LNCS*, pages 131–142. Springer, 2005.

[30] T. Yu and J. Miller. Neutrality and the Evolvability of Boolean Function Landscape. In *Proceedings of the 4th European Conference on Genetic Programming (EuroGP)*, volume 2038 of *LNCS*, pages 204–217. Springer, 2001.