MOVES: A Modular Framework for Hardware Evolution

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

Abstract

In this paper, we present a framework that supports experimenting with evolutionary hardware design. We describe the framework's modules for composing evolutionary optimizers and for setting up, controlling, and analyzing experiments. Two case studies demonstrate the usefulness of the framework: evolution of hash functions and evolution based on pre-engineered circuits.

1 Introduction

In the last years, many authors have presented experiments in which digital hardware has been designed by evolutionary optimization. Perhaps the most popular model for representing a hardware circuit is the Cartesian Genetic Programming (CGP) model [13]. Many variants of this model exist, e.g., [16]. Generally, we can observe a huge variety of proposed hardware representation models, evolutionary operators, and evolutionary optimization strategies – yet alone their manyfold parameter settings. Then, evolvable hardware experiments are very time-consuming. Typically, for a single experiment many simulation runs with different random number seeds and parameter settings are required.

To facilitate evolvable hardware research we have been developing a framework within our MOVES project that enables the rapid setup of experiments, the interactive and offline visualization and analysis of results, and a cluster-accelerated execution. The MOVES framework is easily extensible as it features a highly modular composition with a clear separation of algorithms, representation models, operators, and tools for simulation control and visualization. This allows for a better comparability between different algorithms or simulation setups.

The main contribution of this paper is the presentation of the MOVES framework for experimenting with hardware evolution. We think that such a framework is indispensable for conducting larger, comparative and repetitive experimental studies and, thus, of value to the community. Further, we discuss an experiment to evolve arithmetic circuits based on an initial population including functionally correct, pre-engineered circuits and an experiment to evolve a hashing function for a cache controller.

The paper is organized as follows: In Section 2, we describe our framework's structure and its main modules regarding the implementation of evolutionary optimizers for hardware design. The modules supporting experimentation are presented in Section 3. Section 4 discusses two case study experiments in evolvable hardware that demonstrate the usefulness of our framework. Finally, Section 5 concludes the paper.

2 Composing Evolutionary Optimizers

The goal of the MOVES framework is the separation of the different functionalities required for experimenting with hardware evolution. Consequently, the modules of the framework are divided into two major groups, modules that constitute the evolutionary optimization techniques and modules that serve the experimentation process, including experiment setup and control, statistic analysis, and visualization. The modules of the latter group are described in Section 3.

The modules for the evolutionary optimization techniques are shown in Figure 1 and include the hardware *representation model*, the *evolutionary operators*, and the *evolutionary algorithms*. This separation is suitable for all population-based optimization techniques that apply evolutionary operators, e.g., genetic programming, genetic algorithms, and evolutionary techniques. In the following, we describe the modules together with the currently implemented functionality.

2.1 Hardware Representation Models

The hardware representation is in a sense the most fundamental module as it potentially carries the most dependencies to other modules of the framework. Technically, the representation model is an implementation of the chromosome class. Two different models are currently implemented in the framework:

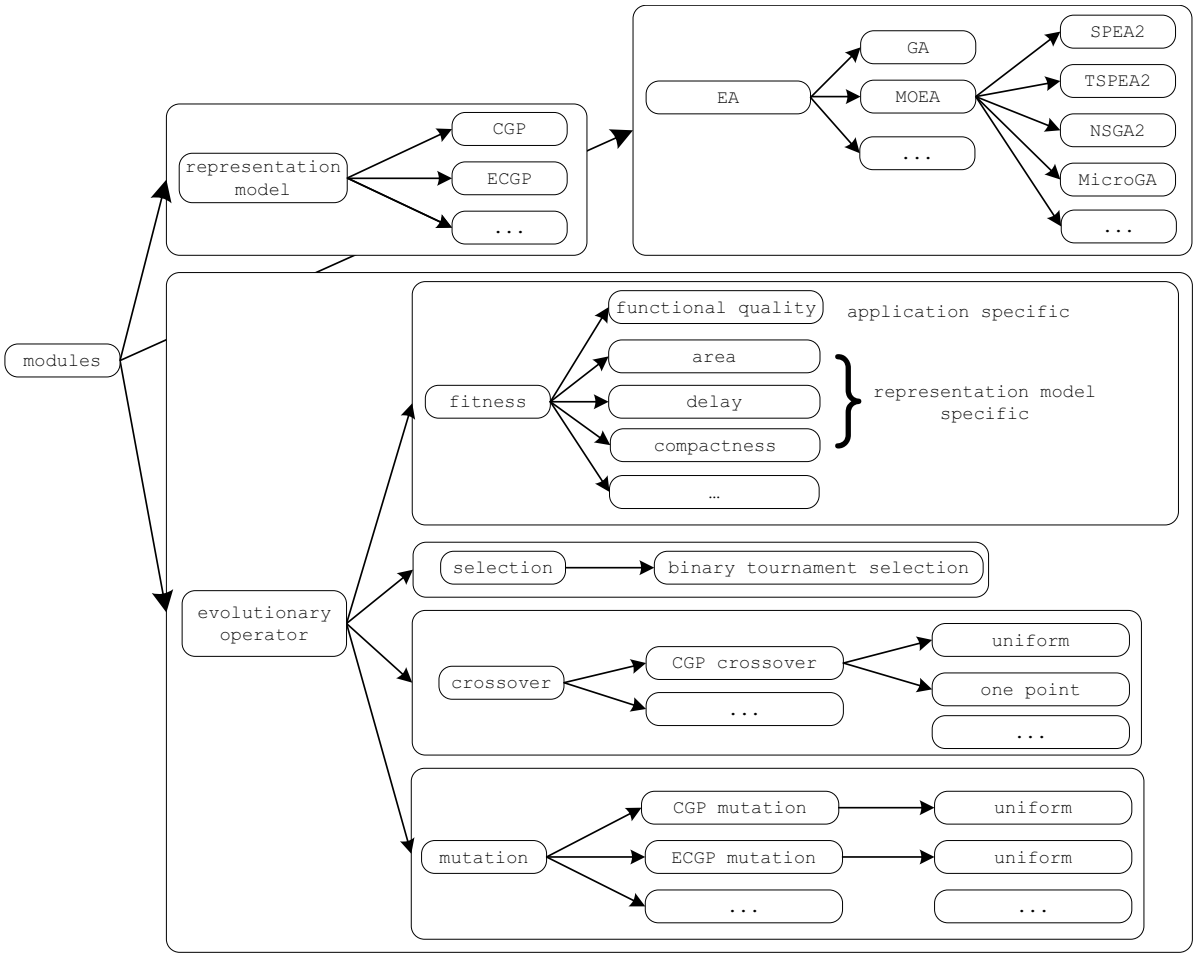

Figure 1. Main modules for composing evolutionary optimizers

- *Cartesian Genetic Programming (CGP)* is the most popular hardware representation model in the context of evolutionary circuit design and was proposed by Miller and Thomson [13]. CGP works on a directed acyclic graph that is spatially constrained by the geometrical structure of a two-dimensional array. Depending on its position in the chromosome, each gene maps to a specific position on that array. The popularity of the CGP model stems from its closeness to real FPGA hardware. CGP models can be mapped rather easily to FPGAs, as technology mapping and placement are already given. Routing is typically left to backend tools, although some approaches encode partial routing as well. We have implemented a highly parameterizable version of the CGP model in our framework. For example, the blocks' logic functions can either be specified by an arbitrary set of gates or by lookup table structures. An example of a CGP individual implementing a hashing function is shown in Figure 2. The main observation we and others, e.g., [11], [2], made by experimenting with the CGP model is that is suffers from two disadvantages. First, the fine-grained computational nodes and interconnects define a large search space leading to the problem of scalability. It takes already a substantial runtime to evolve simple arithmetic functions with small input sizes. Second, the structural properties of the CGP model makes it difficult for the evolutionary operators to automatically extract and work on meaningful substructures.

- *Embedded Cartesian Genetic Programming (ECGP)* is a model first presented by Walker et al. [16]. ECGP extends CGP by the idea of creating larger building blocks out of the basic logic functions. In analogy to biology, ECGP tries to build *organs* out of the basic cells provided by CGP. Related concepts are incremental evolution [15] and automatically defined functions [9]. These approaches are currently considered a promising approach to fight the problem of scalability. The main issue in creating organs is how to identify

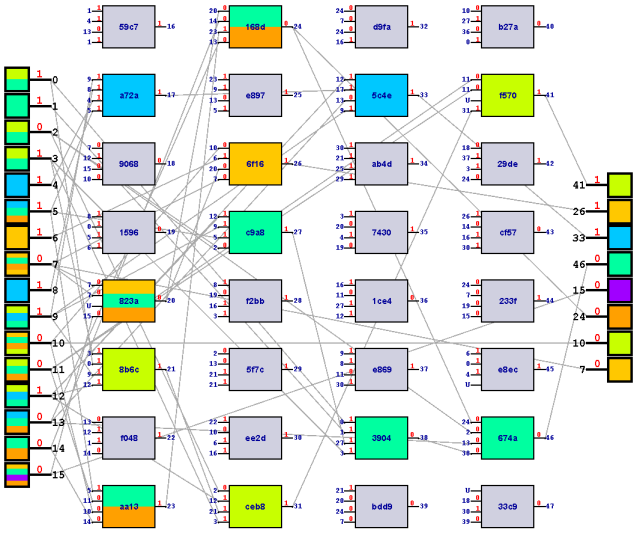

Figure 2. Example of an evolved hashing function on a CGP model (8×4 array, 16 primary inputs, and 8 primary outputs)

suitable blocks. Walker and Miller [16] select blocks randomly. Alternatively, we are experimenting with techniques that take into account the number of generations for which single nodes or sets of nodes persist in the population.

2.2 Evolutionary Operators

Crossover and mutation operators depend strongly on the hardware representation model. In the CGP model, a chromosome is encoded by a column-wise enumeration of genes. A gene is represented by its node function and its connections to predecessor nodes. We have currently implemented two crossover methods, uniform and one point crossover. Uniform crossover selects for every gene of a newly generated individual (child) randomly one of the parents' genes on the corresponding position. In contrast to that, one point crossover creates a child by selecting the first half of the chromosome from one parent and the second part of the chromosome from the other parent. The mutation operator on the CGP model selects one gene with a certain probability and manipulates either its logic function or one of its connections. Following the suggestion in [16], we use only mutation but no crossover for the ECGP model.

Selection operators that rely exclusively on the fitness or objective values are independent of the chromosome representation. We have implemented a standard binary tournament selection scheme that is used, for example, by classic genetic algorithms. Some evolutionary algorithms, especially multi-objective evolutionary algorithms, rely on more complex selection schemes. They might employ breeding and archive populations and use a standard selection scheme only for the breeding population.

The fitness evaluation can include a number of fitness metrics. The functional quality of a hardware circuit is possibly the most often used fitness metrics. Apparently, it is possible to specify a (small) combinational digital function by providing its truth table. Hence, in principle we could define an application-independent functional quality by, e.g., comparing the number of correct output vectors to the overall number of possible test patterns. We did not implement such an evaluation as it is too restricted. First, for functions with a higher number of inputs, it will be not possible to test for all input vectors. Second, many – and in our view the more interesting – functions do not reveal a binary correctness. An example is the hashing function. The functional quality of the hashing function is best measured by its ability to evenly distribute the input keys, which depends on the input data. In our framework, we provide a basic function to compute a circuit's output for a given input vector but foresee a separate and specific evaluation method for each application.

As further fitness metrics, we have implemented area and delay. Area and delay are important characteristics of digital circuits which depend only on the chromosome and the used representation model and not on the specific application. We provide straight-forward estimates for those parameters. The area of an individual can be determined by counting the number of used blocks in the original array. Assume the number of logic blocks used by a circuit c, denoted as $used\_blocks(c)$, is in the range $\{0, \ldots, m\}$. Based on this number, the fitness with respect to area is defined as:

$$area(c) = 1 - \frac{used\_blocks(c)}{m} \qquad (1)$$

A circuit with minimal area gets an area value of 1, a circuit that utilizes all available logic blocks has an area value of 0. The delay is estimated by the depth of the evolved network. Assume the delay of a circuit is in the range $\{0, \ldots, n\}$. Then, the fitness with respect to speed is determined as:

$$speed(c) = 1 - \frac{delay(c)}{n} \qquad (2)$$

The speed equals 1 for the fastest possible circuit and 0 for a circuit with maximal delay.

For more accurate area and delay values, we have coupled the standard Xilinx place and route tools as backend to our fitness evaluation. To this end, we rely on the `JHDL` package [1] which generates an EDIF netlist for the (relatively placed) evolved circuits. The Xilinx design implementation tools are run on this netlist. While we gain accurate fitness values for area and delay that way, evolving hardware circuits becomes very time-consuming.

2.3 Evolutionary Algorithms

Evolutionary algorithms is the last module in our framework. Here, we instantiate a specific representation model and call methods from the other modules. We have currently implemented and tested the following algorithms: GA, SPEA2, TSPEA2, NSGAII, μGA and μGA2. GA is a classic elitism-based single-objective evolutionary algorithm. SPEA2 was presented by Zitzler et al. [18] and is a well-known multi-objective evolutionary algorithm. TSPEA2 is our own variant of SPEA2 that is able to prefer one of the objectives over the remaining ones while still trying to preserve the diversity in the Pareto front [8] [7]. Deb et al. introduced NSGAII [5], a non-dominating sorting multi-objective optimizer. Finally, Coello Coello and Pulido presented the μGA [3] and μGA2 [14] algorithms. Further, the multi-objective optimizer OMOEA2 [17] is already implemented but needs to be tested.

2.4 Implementing new Schemes

The MOVES framework is realized in `Java`. Its modular structure allows to compose a representation model, evolutionary operators, and an evolutionary algorithm to form an evolutionary optimizer. Through its object orientation, the framework can be easily extended. To implement new schemes, the appropriate modules have to be implemented which, mostly, means deriving new classes from existing ones and overwriting methods.

For example, defining a new fitness function is presumably the most frequent extension and requires the implementation of a fitness evaluation method. This can always be done by overwriting the `calculateFitness()` method of the fitness class. However, fitness metrics which can be expressed by a single equation do not need to overload the `calculateFitness()` method. Rather, they are specified in a text file and compiled directly into `Java` byte code at the start of the simulation run using the `JEL` library [12].

3 Experimentation Support and Tools

In this section, we describe the modules of the MOVES framework that deal with the setup, control, visualization, analysis, and distribution of experiments. The overall framework is developed in `Java` and thus platform-independent. The framework uses several external libraries such as `jgrapht` for efficient graph representation and manipulation, `jgraph` for graph visualization, and for the online visualization of the optimization progress `jfreechart`.

3.1 Creating and Controlling Experiments

An experiment is defined by creating two configuration files; one specifies the evolutionary optimizer and another one the experiment setup. The evolutionary optimizer configuration includes the chosen representation model, evolutionary operators, and evolutionary algorithm. The experiment setup configuration comprises termination conditions, visualization settings, and the logging frequency. The configuration can be provided as regular text files or, alternatively, be entered via the framework's graphical user interface.

Experiments can be run interactively or in batch mode. In the interactive mode, the user can pause, resume, or stop the simulation at any time. Generally, the parameters controlling the experiment setup can be modified during the experiment. For example, the experiment can be executed step-wise or continuously until some of the termination conditions are reached. The user is free to switch between these modes at any time. It is also possible to save the current search state and reload it later on to analyze it. These features are extremely useful to debug and verify new representation models and their corresponding operators, and to tune the parameters of the evolutionary algorithm. The batch mode is used for an unattended simulation. Statistical data can be gathered during the experiment and exported to a text file for later processing.

Experiments are generally logged in a textual format. The log contains algorithmic parameters and the important information about the search process. To this end, every instance of an operator and an evolutionary algorithm can be associated with an output stream to realize a specific data logging. By default, after each generation a complete report including the parameters used and a textual presentation of the best individual is logged.

We have further implemented a checkpointing mechanism. Checkpoints can be scheduled every n generations or after every n seconds. When a checkpoint is reached, the framework saves a snapshot of the complete search state. Checkpointing serves two goals: First, we are able to restart the experiment after a breakdown. Second, by an offline analysis of the checkpoints the optimization run can be completely analyzed.

3.2 Visualization and Analysis

There are two visualization tools, the visualization of the representation model (chromosome) and the visualization of the evolutionary optimization process. For visualizing the representation, the representation model has to implement the `Java swing` component `JPanel`. As an important feature, the user can use the chromosome represen-

Figure 3. Example screen shots from our framework: graphical representation of a specific individual on the CGP model (background window), development of the fitness over time (foreground-left window), and current Pareto fronts (foreground-right window)

tation to modify the viewed data, e.g., to manipulate input data or even change the circuit's structure and function. We have already implemented this functionality for the CGP and ECGP models. If this class is missing for a specific representation model, a textual representation of the chromosome is drawn.

The standard visualization of the evolutionary optimization process displays the progress of the best fitness and the population's average fitness over the generations. This is most useful for single-objective optimizers. In experiments with multi-objective optimizers we can display the progress of all individual fitness values, and the two-dimensional projections of the Pareto fronts.

The visualizations can be done either during an experiment run or offline using the `Statistic-Viewer` and `Chromosome-Viewer` tools on previously saved experiment log streams.

3.3 Distributed Simulation

For an evolvable hardware experiment, usually dozens of simulation runs are required. As evolutionary algorithms are stochastic optimization methods, several runs with different seeds for the random number generators are conducted. Often, parameters need to be varied in given ranges, each setting requiring a number of simulation experiments. Such experiments can be tedious to set up and take a very long runtime. The single experiments, however, are independent of each other and amenable to parallel execution.

The MOVES framework is able to automatically create a set of experiments (experiment configuration files) where parameters are varied in specified intervals and with defined step sizes, and to execute all simulations as batch jobs on a compute cluster. We employ the grid software `Condor` [6] that distributes the jobs on the computing nodes in the cluster, monitors the node's activities, and relocates the jobs if it becomes necessary. `Condor` itself does not support the migration of multithreaded `Java` processes. Hence, we leverage on our checkpointing mechanism to restart a process in case `Condor` decides to migrate the process to a different machine. Before submitting an experiment to `Condor`, our framework composes all experiment class files and the used libraries into a single `jar` and creates the according submission file. Upon termination of the experiment, the standard input, output and error streams, as well as the checkpointing and log files are transferred to the `Condor` server. There, the data can be viewed and analyzed. To keep track of the progress, the MOVES framework includes a network monitor module which displays the status of executed experiments.

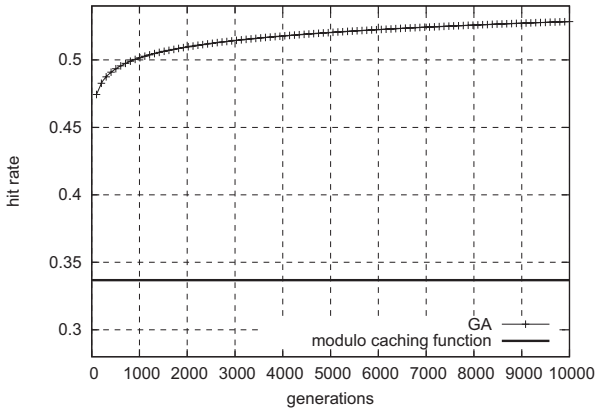

Figure 4. Evolving a hashing function. The fitness is given by the hit rate of the cache.

4 Evolvable Hardware Case Studies

4.1 Hashing Functions

In this first case study, we report on experiments with evolving hashing functions. The hashing function is a prominent example for the group of combinational digital functions that have no binary correctness. The functional quality of the hashing function is measured by its ability to evenly distribute the input keys. In contrast to the often investigated arithmetic functions, a hashing function's performance is strongly dependent on the input data, i.e., the distribution of keys. The offline optimization of a hashing function to a given key distribution might be of interest on its own right. However, the hashing function is further amenable to online evolution, one of the techniques considered for the construction of self-adapting systems.

Experiments on evolving a hashing function on the gate level have been presented by Tettamanzi et al. [4]. To be able to compare their experiments with ours, we used the same CGP parameters: the number of columns in the array (n_c) equals 8, the number of rows (n_r) equals 8, the number of primary inputs (n_i) is 16, the number of primary outputs (n_o) is 8, the number of columns a connection can reach back to the primary inputs (l) is also set to 8, and the number of block inputs (n_n) is 4. For the set of possible node functions, we experimented with two versions: allowing all possible node functions of 4 inputs (as Tettamanzi et al. did), and restricting the node functions to the set `AND, ONE, OR, XOR, NULL, NAND, NOT, NOR` and `XNOR`. We could not observe any substantial differences between these variants, neither in functional quality nor in speed of convergence. Tettamanzi et al. restricted wires to connect only to logic blocks in the same row - a constraint we have relaxed in our experiment.

The problem statement is as follows: Find a function $h : \mathbb{B}^{16} \rightarrow \mathbb{B}^8$ which maps a set M of 2^{12} keys (out of 2^{16} possible keys) to a set N of 2^8 indices in the most uniform way possible. The fitness function is defined as:

$$f(h) = \frac{1}{1 + \frac{1}{|N|}\sum_{i=1}^{|N|}(|\{j|j \in M,\ h(j) = i\}| - \frac{|M|}{|N|})^2}$$

Tettamanzi et al. [4] evolved the best individual with a fitness value of 0.097785 after 257 generations. Due to our relaxed routing constraints, we could achieve improved results. The single-objective GA reached easily an average fitness beyond 0.1. After 257 generations, the best individual showed a fitness of 0.116469. The experiments with hashing functions have been conducted on a cluster of 18 nodes, each one being a P4-3GHz with 2GB RAM.

Further, we have conducted initial experiments in evolving cache controllers. As an example, using the `MIPS` simulator `SPIM` [10] we have generated a trace of programm counter addresses for a quick sort algorithm. To sort a given test array of integers, the program executes 174603 instructions which are fetched from 96 different addresses. Now we have considered a very simple instruction cache organization with 16 cache lines and one word per line, and tried to evolve a hashing function for a cache controller optimized to quick sort. The functional quality was defined as follows:

$$f = \frac{\text{\#hits}}{\text{\#executed instructions}}$$

The hashing function has been evolved on a CGP model with $n_c = n_r = l = 8$ and $n_n = 4$. The number of inputs was restricted to $n_i = 8$, and the number of outputs had to be $n_o = 4$. We allowed all possible node functions for the 4-input nodes. Figure 4 shows the result of the experiment. The evolved hashing function achieves a hit rate of 0.55 after 10'000 generations, whereas a direct mapped cache would show a hit rate of 0.33. In terms of cache hits, our hash function leads to 96212 hits compared to the 58804 hits for the direct mapped cache. We have to note that this comparison is not fair, as in a real MIPS-based system, much larger caches with multi-word blocks and higher degrees of associativity would be used. Further, we have not yet worked out a complete concept for the implementation of an evolvable hardware cache. However, cache controllers might be an interesting application area for evolutionary hardware.

4.2 Evolution based on Pre-engineered Circuits

With the second case study, we present two ideas for evolving hardware. The first is to use a multi-objective evolutionary optimizer to evolve an approximated Pareto front

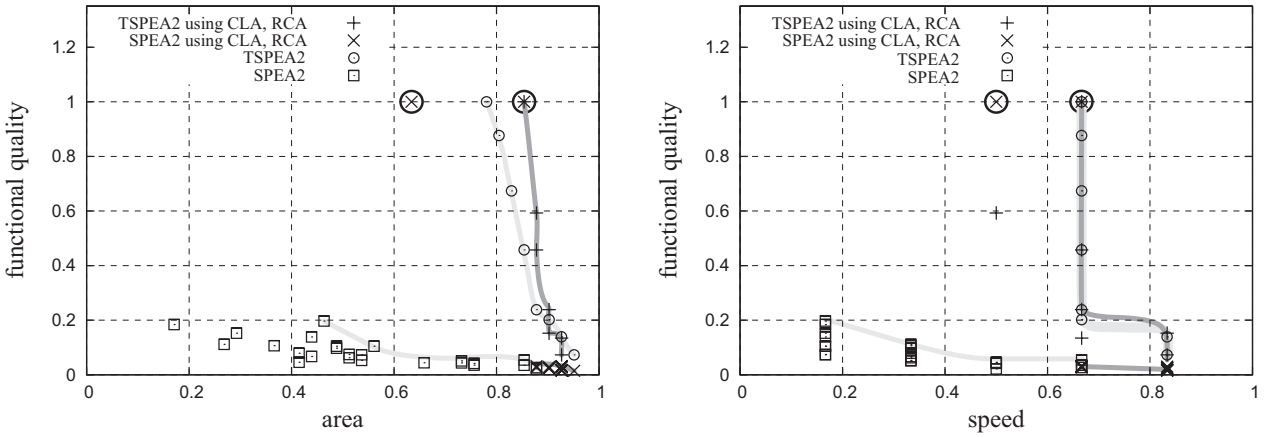

Figure 5. Evolving 3×3 adders with SPEA2 and TSPEA2. Two-dimensional projections of the three-dimensional Pareto-front are plotted. The objectives on the x-axes are calculated as specified in Section 2.2. The pre-engineered carry look ahead adder (CLA) and ripple carry adder (RCA) are marked by bold circles.

of functionally good circuits with small area and short delay (high speed). In previous work, we have experimented with SPEA2 [18], a widely-used multi-objective evolutionary algorithm, and proposed TSPEA2 [7]. TSPEA2 is a variant of SPEA2 that is able to prefer one of the objectives over the remaining ones while still trying to preserve the diversity in the Pareto front.

The second idea looked at in this case study is to start the evolution of arithmetic circuits with an initial population including classically engineered circuits. There exists a vast knowledge on how to design arithmetic circuits of all kinds. Despite that fact, many authors strived for evolving such circuits with less gates. However, it turns out that evolving functionally correct arithmetic circuits is a challenge in the first place.

We have chosen a 3×3 adder to examine the idea of using pre-engineered circuits. We have constructed an initial population with 5% ripple carry adders (RCA), 5% carry-look-ahead adders (CLA), and 90% randomly generated circuits. Then, we have tried to evolve Pareto fronts with both the SPEA2 and the TSPEA2 algorithm on a CGP model with $n_c = 5$, $n_r = 8$, $n_i = 6$, $n_o = 4$, $n_n = 4$, and $l = 5$. The node functions could be arbitrary functions of 4 inputs. The population size has been set to 100. The experiments have been run for 100.000 generations.

Figure 5 shows typical Pareto fronts for SPEA2 and TSPEA2, each one with pre-engineered circuits in the initial population and with a standard random initial population. The left-hand side of Figure 5 presents the functional quality over the area; the right-hand side displays the functional quality over the speed. A functional quality of 1 denotes a correct adder. An area of 1 and a speed of 1 are the minimal values.

The following observations can be made: TSPEA2 evolves circuits with much better functional quality than SPEA2 by preferring functional quality over area and speed. Moreover, in this experiment all circuits evolved by SPEA2 are dominated by some circuits evolved by TSPEA2. The second observation is that evolving with pre-engineered circuits results only in small improvements. TSPEA2 could evolve circuits with better area, but identical speed parameters. There are several reasons for this. The speed estimates are rather coarse (see Section 2) which makes a differentiation difficult, especially for small CGP arrays. Here, using the Xilinx backend tools to route the circuits and get more accurate delays would be useful. Then, the crossover operator for the CGP model does not respect the spatial arrangement of genes. Useful substructures are not recognized as such and are likely to be destroyed. Based on these experiments we conclude the insufficiency of a blind crossover operator on the CGP representation model. We plan to redo this experiment on the ECGP model.

5 Conclusion

In this paper we presented MOVES, a modularized framework for experimenting with evolutionary algorithms for hardware design. We described the main modules of the framework, including modules required to implement evolutionary optimizers and modules required to set up, control, analyze and distribute experiments. The usefulness of such a framework has been demonstrated by two case studies, evolving hashing functions and evolving adders based on pre-engineered circuits.

6 Acknowledgement

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

References

[1] P. Bellows and B. Hutchings. JHDL - An HDL for Reconfigurable Systems. In K. L. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 175–184, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[2] X. Cai, S. L. Smith, and A. M. Tyrrell. Positional Independence and Recombination in Cartesian Genetic Programming. In *Genetic Programming, 9th European Conference, EuroGP 2006, Budapest, Hungary, April 10-12, 2006, Proceedings*, volume 3905 of *Lecture Notes in Computer Science*, pages 351–360. Springer, 2006.

[3] C. A. C. Coello and G. T. Pulido. A Micro-Genetic Algorithm for Multiobjective Optimization. In *Evolutionary Multi-Criterion Optimization: First International Conference, EMO 2001*, page 126, Zurich, Switzerland, 2001. Springer.

[4] E. Damiani, V. Liberali, and A. Tettamanzi. Evolutionary design of hashing function circuits using an fpga. In *International Conference on Evolvable Systems (ICES)*, pages 36–46. Springer, 1998.

[5] K. Deb, S. Agrawal, A. Pratab, and T. Meyarivan. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. In *Proceedings of the Parallel Problem Solving from Nature VI Conference*, pages 849–858, Paris, France, 2000. Springer.

[6] D. H. J. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing Among Workstation Clusters. In *Future Gener. Comput. Syst.*, volume 12, pages 53–65, Amsterdam, The Netherlands, 1996. Elsevier Science Publishers B. V.

[7] P. Kaufmann and M. Platzner. Multi-objective Intrinsic Hardware Evolution. In *Proceedings of the 2006 MAPLD International Conference*, Washington D.C., USA, September 2006. To appear.

[8] P. Kaufmann and M. Platzner. Toward Self-adaptive Embedded Systems: Multi-objective Hardware Evolution. In P. Lukowicz, L. Thiele, and G. Tröster, editors, *Proceedings of the 20th International Conference on Architecture of Computing Systems (ARCS 2007)*, volume 4415 of *LNCS*, pages 199–208. Springer, MAR 2007.

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

[10] J. Larus. SPIM - A MIPS32 Simulator.

[11] Liu, Rui and Zeng, Sang-you and Ding, Lixin and Kang, Lishan and Li, Hui and Chen, Yuping and Liu, Yong and Han, Yueping. An Efficient Multi-Objective Evolutionary Algorithm for Combinational Circuit Design. In *First NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 215–221. IEEE, 2006.

[12] K. L. Metlov. JEL - Library for Evaluating a Simple Single Line Expressions in Java., 2006.

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

[14] G. T. Pulido and C. A. C. Coello. *The Micro Genetic Algorithm 2: Towards Online Adaptation in Evolutionary Multiobjective Optimization*. Springer, 2003.

[15] J. Torresen. Two-Step Incremental Evolution of a Prosthetic Hand Controller Based on Digital Logic Gates. In *Proceedings of Evolvable Systems: From Biology to Hardware: 4th International Conference on Evolvable Hardware (ICES 2001)*, volume 2210 of *Lecture Notes in Computer Science*, Tokyo, Japan, 3–5 Oct. 2001. Springer.

[16] J. A. Walker and J. F. Miller. Improving the Evolvability of Digital Multipliers Using Embedded Cartesian Genetic Programming and Product Reduction. In *Evolvable Systems: From Biology to Hardware, 6th International Conference, ICES 2005, Sitges, Spain, September 12-14, 2005*, volume 3637 of *Lecture Notes in Computer Science*, pages 131–142. Springer, 2005.

[17] S. Zeng, S. Yao, L. Kang, and Y. Liu. An Efficient Multi-objective Evolutionary Algorithm: OMOEA-II, 2005.

[18] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Swiss Federal Institute of Technology, Gloriastrasse 35, CH-8092 Zurich, Switzerland, 2001.