
Evolvable Hardware using State-machines

Magnus Ekman

Department of Computer Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
mekman@ce.chalmers.se

Peter Nordin

Department of Physical Resource Theory
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
nordin@fy.chalmers.se

Abstract

In general, approaches to genetic programming are more efficient if the implementation is close to hardware. For instance, evolution of binary machine code is very efficient time-, space- and energy-wise. Instead of evolving machine code for a fixed hardware it is possible to evolve the hardware. This paper describes an approach evaluated by experiments that test evolution of hardware at a high abstraction level. The method is evaluated on some small sample control problems. The evolution is performed extrinsically and produces VHDL-code, which can be synthesized and implemented in an FPGA. Further, a method is suggested that can be tested on chip but without the computational expensive place and route phase. This will speed up the evolution significantly.

1 INTRODUCTION

In general, approaches to genetic programming are more efficient if the implementation is close to hardware. In AIM-GP [1] binary machine code is directly evolved resulting in a speed up of several orders of magnitude, while also enabling compact and power efficient implementation on low-end computer architectures. The next logical step for increased efficiency is to move below the level of processors into reconfigurable logic chips, such as FPGAs. This is quite straight forward as long as the evolution and fitness function is limited to logical functions [2]. However, many domains require evolution of functionality composed of blocks at a higher level, for example arithmetic blocks in numerical domains. Hardware evolution can thus be performed at various abstraction

levels. Work has been done at low levels by Thompson et al. [3] who have evolved configuration strings for an FPGA. One problem with this is that the configuration strings for the newer bigger FPGAs are not public, but Levi et al. have presented a way to solve this problem [4]. Murakawa et al. have investigated hardware evolution at function level [5], using more complex functions instead of working with primitive two-input gates.

The variant that is suggested in this paper is more abstract than function level hardware evolution. It is based on a fixed structure of state-machines and is at a lower level than a register machine but still at a higher level than pure logic functions. It is interesting to note that the approach evolves a hardware variant of one of the first paradigms for evolution of functionality in the early work of Evolutionary Programming [6]. A program called VHDLGP has been implemented that evolves a construction described in VHDL-code, which is tested extrinsically. The program has been tested on some small problems and the method seems promising.

The testing phase is a problem when hardware evolution is performed at a high abstraction level. If the test is done with an FPGA, a time-consuming place and route phase that maps the construction to the physical device is needed. An alternative is to simulate the construction. In the last section of the paper an improvement to the state-machine structure is presented which enables intrinsic evolution in an FPGA without a place and route phase.

2 MACHINE-LEVEL APPROACH

The method evolves VHDL-code using a custom structure of hardware. This structure consists of several state machines of various sizes that co-operate to solve the problem. The reason to have several machines instead of one big complicated one is that no appropriate

way to mix two machines and still keep the properties of both seems to exist. We thus use several state-machines to make crossover easier to implement — an alternative approach would be to evaluate the system without crossover using only mutation, e.g. the way state-machines are treated in early work of Evolutionary Programming [6]. The structure presented below seemed like a promising compromise:

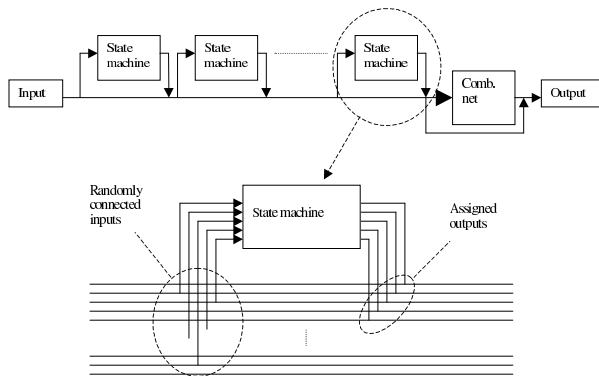


Figure 1: Structure of the evolved construction

It is built from several state machines that all are synchronous. They will only be of the type Moore or pure Moore machines [7]. If it is a Moore machine the output values will be evolved, and if it is a pure Moore machine the state bits will be presented on the output of the machine. However, the combinatorial net on the output of the entire structure makes Mealy structures possible. Every machine, as well as the input to the system, is assigned a constant number of bits on the bus that it can write to. If the state machine does not have so many outputs, the unassigned bus pins will be assigned logical zeros. The inputs to the machines are also connected to the bus but they are not assigned to predetermined pins. These connections are evolved. Depending on the settings they can be connected to any pins on the bus or just to bits located close to the output pins of the machine. The combinatorial net on the output can be connected to any of the bits on the bus. The output from the structure can be any of the bits on the bus or any of the outputs from the combinatorial net.

With this structure each machine will get its own task, i.e. to present good outputs on its bits on the bus. The machine can do this with the help of outputs from other machines, inputs to the structure and different state transitions.

2.1 FITNESS FUNCTION AND SYSTEM ENVIRONMENT

It is possible to supply a system environment, in which the evolved construction is inserted. The task of the system environment is to supply stimuli and handle outputs from the evolved construction. The system environment also includes the fitness function. The system environment and fitness function is modeled in VHDL-code since this is probably how it will be represented in the implementation once the evolution is finished. The world outside the FPGA is modeled as stimuli to the FPGA pins.

2.2 GENETIC OPERATORS

The genetic operators that are used are crossover, mutation and duplication. The crossover in this structure exchanges state machines between two individuals and the combinatorial net is exchanged or mixed between two individuals. Each state machine is assigned its bits on the bus and this is not altered. This is necessary to keep some properties of the parents. The combinatorial net on the output is either selected from one of the parents or it is a mixture of the two nets from the parents. The mixture of the net is done by taking some rows from the truth table of the net from the first individual and some rows from the second individual. A picture of the crossover is shown below.

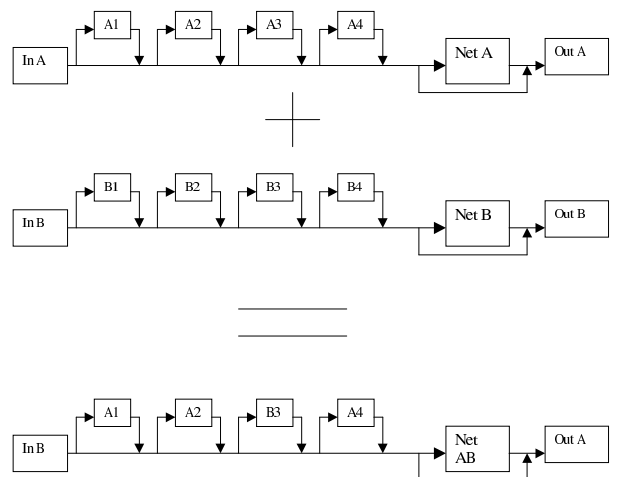


Figure 2: The crossover operator

The mutations that are used are change in a state-transition, introduction of a new state-transition, deletion of a state-transition, change of output in a state and change of the combinatorial net on the output. The only gene duplication duplicates a state machine and replaces another machine.

2.3 EVOLUTIONARY ALGORITHMS

Two different evolutionary algorithms have been tested. The first one is a steady state tournament selection method with a tournament size of four. The algorithm keeps a population of individuals. A subset of four individuals are randomly selected and sorted according to their fitness values. The two worst individuals are removed from the set. The other two are combined with the help of genetic operators to generate two new individuals that are inserted into the set.

The other algorithm is a traditional evolutionary algorithm [8] that tests all the individuals and then selects parents randomly but with higher probability if the individual has a good fitness value. These parents create a completely new population, so elitism is not used. However, this method is not thoroughly evaluated but may have some benefits in that it is easier to analyze in a theoretical framework.

2.4 TEST PROBLEMS AND RESULTS

One test problem that the program was applied to, was the problem to evolve a multiplier. The evolved system was specified to have two 4-bit inputs and one 8-bit output. Some different numbers were applied to the inputs, and the output was read after a number of clock cycles. The fitness function is described by the formula below.

$$f(x) + \sum_{i=1}^n |y_i - x_i| . \quad (1)$$

Here x represents the output from the structure and y the correct value. The term $f(x)$ punishes individuals that have constant outputs.

VHDLGP almost solved the problem but it did not evolve a general multiplier. The construction evolved was more like a lookup table. This is not very surprising since VHDLGP does not have any information about how binary coded numbers and binary arithmetic work. It is a too big step to take to evolve a multiplier from scratch. A more natural way that could work would be to try to evolve an adder first, and then try to evolve a multiplier using this construction.

Another test problem was to evolve an IR-detector that was to be used in another project. A system environment was written that presented a 10 kHz signal with random intervals. The reason that the 10 kHz signal was present in random intervals was to avoid that

the construction could learn when it should present a high signal on the output without using the input signal. This experiment was a little similar to the experiment done by Thompson et al. but they had two signals with different frequencies that the construction should be able to distinguish from each other. Another difference was the fitness function. In Thompson's experiment the fitness function integrated the output signal with an analogue construction. In this experiment the output signal was read on discrete occasions. The time between readings, was not always the same. The mathematical function that was used to score the constructions is shown below

$$128 + f(x) + \sum_{i=1}^n |y_i - x_i| . \quad (2)$$

Here x represents the output from the structure and y the correct value. These values are just one-bit values. The signal is present or not. The term $f(x)$ punishes individuals that have constant outputs and the constant (128) biases the value so it falls within the range of 0 to 255. Before the fitness function was adjusted (see below) n was 40.

The evolution was successful. A construction was evolved that made no mistakes at all, but it shall be noticed that the only test was to present a 10 kHz signal or no signal at all. No tests were done with signals of other frequencies but there is no reason to expect any major problems with evolving a construction that can distinguish between different frequencies. One thing should be noticed. The construction did not give a steady logic level on the output. The output toggled all the time but the phase was adjusted so that when the signal was read, it always showed the correct value. The data was caught on the positive edge of the clock, and at that time there was always valid data on the output. This was not a problem in this project since the data was only to be used when it was read. However, it could probably have been avoided with a different fitness function. On the next page there is a graph of the learning process. The upper line is the average fitness value and the lower line is the best fitness value. The construction was evolved with the steady state algorithm with a population of 16 individuals. The fitness function was adjusted by changing n to 70 after about 200 tournaments, which explains the rise in average fitness after two thirds of the diagram. The reason to adjust the fitness function was that the evolved constructions scored as well as possible but did not work perfectly when tested on another set of inputs. After about 300 tournaments the evolution was stopped since the constructions were as good as they

could be.

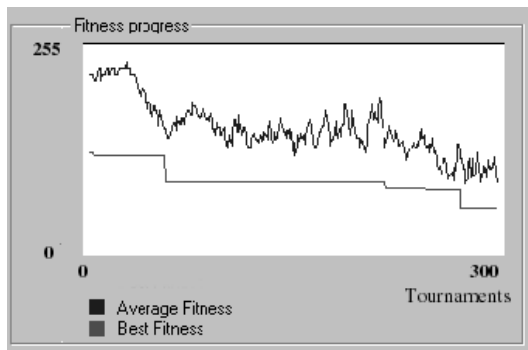


Figure 3: Diagram of the fitness value

3 FUTURE WORK

A way of speeding up the tests would be to build a structure of reconfigurable state machines in hardware. This could be done by keeping the state transition tables in RAM [3] and connecting the inputs to the bus with the help of multiplexers. The combinatorial net on the output could also be coded in a RAM and the outputs from the entire structure could be connected by means of multiplexers. The downloading of the configuration would be fast and the test could be done intrinsically. It is also possible to include predefined units known to be good to partly solve the problem. A possible block scheme for a test circuit is shown below.

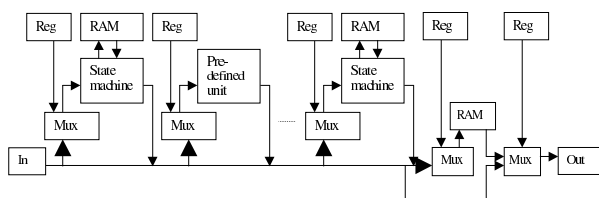


Figure 4: Suggested structure that can be tested intrinsically

The FPGAs on the market today contain block-RAMs and are quite big, which makes it possible to implement this test structure in an FPGA. In this way the evolution can take place in the FPGA without any place and routing during the test phase. Developing and implementing this test structure is a natural way of continuing the work done in this project.

4 SUMMARY AND CONCLUSIONS

The method has proved that it works on proof of concept problems. The solution to the signal detector

was interesting since it did not present a steady output signal but still solved the problem. The learning process takes rather long time due to the simulation, but once the suggested test structure is implemented it will take less time and the method can be applied to larger problems.

5 ACKNOWLEDGEMENTS

This work was done at TietoEnator Embedded Tech AB in Göteborg. We would like to express our gratitude for their support.

References

- [1] Nordin P (1997). Evolutionary Program Induction of Binary Machine Code and its Applications. *Kreihl Verlag*.
- [2] John R. Koza, David Andre, Forrest H Bennett III and Martin Keane (199). *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman
- [3] Thompson A., Layzell P., and Zebulum R. S (1999). Explorations in design space: Unconventional electronics design through artificial evolution. *IEEE Trans. Evol. Comp.*, 3(3):167-196.
- [4] Levi D. and Guccione S. A (1999). GeneticFPGA: A Java-based Tool for Evolving Stable Circuits. In John Schewel, et.al., editors, *Reconfigurable Technology: FPGAs for Computing and Applications*, Proc. SPIE 3844, pages 87-92, Bellingham, WA.
- [5] Murakawa M., Yoshizawa S, Kajitani I, Furuya T, Iwata M, And Higuchi T. Hardware evolution at function level. In *Proc. of the Fifth International Conference on Parallel Problem Solving from Nature*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 1996.
- [6] Fogel, L., Owens, A., and Walsh, M (1965). Artificial intelligence through a simulation of evolution. In Maxfield, M., Callahan, A., and Fogel, L. editors, *Biophysics and Cybernetic Systems*, pages 131-155.
- [7] Fisher, P.D., and Wu, S.-F., Race-Free State Assignments for Synthesizing Large-Scale Asynchronous Sequential Logic Circuits. *IEEE Trans. on Computers*, Vol 42, No 9, pp 1025-1034, Sept 1993.
- [8] Bentley P. J. *Evolutionary Design By Computers*. Morgan Kaufmann Publishers, Inc. San Francisco, California.