GAME-HDL: Implementation of Evolutionary Algorithms using Hardware Description Languages

Rolf Drechsler Nicole Drechsler

Institute of Computer Science
University of Bremen
28359 Bremen, Germany
Email: drechsle@informatik.uni-bremen.de

Abstract. Evolutionary Algorithms (EAs) have been proposed as a very powerful heuristic optimization technique to solve complex problems. Many case studies have shown that they work very efficient on a large set of problems, but in general the high qualities can only be obtained by high run time costs. In the past several approaches based on parallel implementations have been studied to speed up EAs. In this paper we present a technique for the implementation of EAs in hardware based on a the concept of reusable modules. These modules are described in a Hardware Description Language (HDL). The resulting "hardware EA" can be directly synthesized and mapped to Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs). This approach finds direct application in signal processing, where hardware implementations are often needed to meet the run time requirements of a real-time system. In our prototype implementation we used VHDL and synthesized an EA for solving the OneMax problem. Simulation results show the feasibility of the approach. Due to the use of a standard HDL, the components can be reused in the form of a library.

Keywords: evolutionary algorithm, hardware description language, VHDL, hardware realization, implementation, reuse

1. Introduction

In the last 10 years evolutionary techniques have been very successful to a large set of problems in very different domains, like e.g. circuit design [1,2,3,4]. Typically, the results obtained by Evolutionary Algorithms (EAs) can be characterized as being of very high quality, but the run time needed to compute the solutions is high compared to other optimization techniques, like hill climbing or gradient descendent. This is mainly due to the simulation-based approach that is used in EAs. Most of the computation time on complex problems is usually consumed by evaluation of the fitness function. The time needed for the overall algorithm, the selection and the application of the evolutionary operators are negligible.

To reduce the run time, in the past mainly two solutions have been proposed. The first one is based on parallel implementations [5]. Since there is usually no close interaction needed between the different elements of a population, the complete problem can easily be distributed over several hardware platforms. Many promising applications have been reported. A speed-up of close to linear (and in some cases even super-linear) can be achieved. But still the fitness functions are evaluated in software.

A very promising alternative is to implement the EA directly in hardware [6]. Hardware implementations are often several orders of magnitude faster than an equivalent software realization. The main drawback of this approach is the missing flexibility and the difficulty of the implementation, since this requires detailed knowledge about hardware development. To meet today's requirements to allow for fast prototyping more robust structures are needed that allow an inexpensive development while keeping the advantages of the underlying methods. Especially in the field of signal processing the demands regarding execution speed can often only be met by hardware implementations.

In this paper we present a dynamic hardware concept based on the use of standardized Hardware Description Languages (HDLs). All EA components, like the population (including the initialization), the selection mechanism and the fitness evaluation, are directly implemented as hardware modules that can be synthesized on a target technology. This can be an ASIC or an FPGA. For our implementation we used VHDL [7,8], since it is widely used and a large set of commercial tools exist. The VHDL code describes the different components in a structured way that allows for an easy reuse. This becomes the key of today's successful hardware projects, since without a concise reuse-methodology the complexity cannot be handled [9,10]. The overall structure of the system is closely related to the concept in [11], where GAME (=Genetic Algorithm Managing Environment) has been developed. Since the same modular structure can be found in the hardware counterpart, the library of modules is denoted as GAME-HDL. As a first case study we tested the components on OneMax problem, i.e. maximizing the number of ones in a bit-string. An experimental study including simulation results is given to show the flow of the hardware implementation.

2. Preliminaries

In this section we briefly review the notation and definitions used to make the paper self-contained. First, the concept of EAs is described (see also [1]) and the components used later are given. Then, an introduction to VHDL – the HDL used throughout the paper – is provided.

2.1. Evolutionary Algorithms

We assume that the reader is familiar with the concepts of simulated evolution for computer optimization problems. Since the late 70s, several concepts using simulated evolution have been proposed. Among them are e.g. *Genetic Algorithms* (GAs) [12], *Evolution Strategy* (ES) [13,14] and *Genetic Programming* (GP) [15]. The different concepts mainly differ in the form of representation of the individuals in a population and in the operators applied, while the overall algorithmic flow is very similar. In the following, we do not distinguish between these "pure'" concepts and use the term *Evolutionary Algorithm* as the union of all these techniques.

We do not discuss the components of EAs, i.e. representation, objective function, selection method, initialization of population, and evolutionary operators. (For more details see e.g. [16].) Based on these components the overall structure and information flow of a "classical" EA works as follows:

- Initially a random population is generated.
- The evolutionary operators reproduction, crossover, and mutation are applied to some elements. The elements are chosen according to the selection method.
- The algorithm stops if a termination criterion is fulfilled, e.g. if no improvement is obtained for a given number of iterations.

The EA depends on several parameter settings, e.g. population size, reproduction probability, crossover probability, and mutation probability. A sketch of a "classical" EA is given below.

```
evolutionary_algorithm (problem instance):
  initialize_population ;
  calculate_fitness ;
  do
    apply_operators_with_corresponding_probabilities ;
    calculate_fitness ;
    update_population ;
  while (not terminal case) ;
  return best_element ;
```

2.2 Hardware Description Languages

HDLs are programming languages comparable to languages used in the software domain, like C++ or JAVA, but the compilation process generates a chip instead of an executable program. For this, HDLs contain – in addition to constructs to describe the logic behavior, like if-then-else and loops – also commands to specify implementation specific properties, like timing. The need for standardized HDLs comes from several different aspects. Some of the most important ones are:

- The ever increasing design complexity of today's designs can only be handled by structured languages, since the circuits consist of up to several million transistors and are described by several thousand lines of code.
- Components developed once can be reused several times. In ASIC projects often more than 90% of the modules are reused.
- In design flows several tools have to interact, like the synthesis and simulation tools. The code in the HDL is used as an exchange format.

- The circuits can be developed independent of technology or tool provider.

Typical HDLs used in practice today are VHDL and Verilog. Recently, also languages have been proposed that are closer to software descriptions, since this often allows to describe hardware and software components in the same environment and enables a late decision which parts become software and which are realized by hardware (*hardware-software-co-design*).

For our implementation we used VHDL, since it is the most common language on the European market. Furthermore, several commercial tools are available. Within this article, a complete introduction to VHDL cannot be provided. Instead, a short example is given that explains the main components.

For each module the interface has to be declared. This is done using an `ENTITY`. It declares the input and output signals of each module and their types.

```
ENTITY AND_OR IS
PORT ( x1, x2, x3 : IN BIT;
                f : OUT BIT ) ;
END AND_OR ;
```

The functional behaviour is described by the `ARCHITECTURE`, that has to be provided for each `ENTITY`.

```
ARCHITECTURE function OF AND_OR IS
BEGIN
  f <= x1 AND (x2 OR x3) ;
END function ;
```

In our example, the module has three inputs and one output. The output `f` is computed by Boolean operators. Of course, modern HDLs also provide more complex operations and data types, like e.g. addition and multiplication of bit-vectors. For each `ENTITY` at least one `ARCHITECTURE` has to exist. If there are several, e.g. some components may be optimised for area while others are better regarding delay, an assignment has to be done using a `CONFIGURATION` command. Further features are:

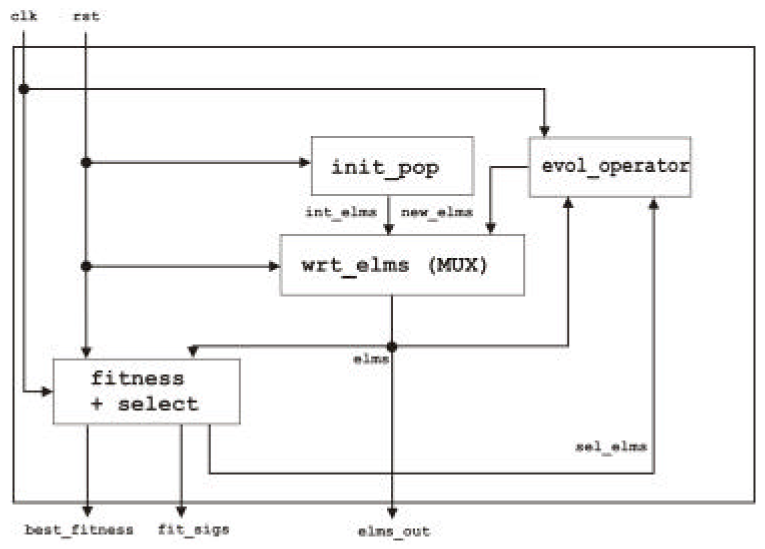
- To simplify reuse, corresponding to libraries in other programming languages, frequently used data types and modules can be grouped to build a `PACKAGE`.
- VHDL furthermore supports different description methods, i.e. structural, behaviour and data flow.
- Signal delays can be modelled.
- Availability of complex data types, like integer, real, or array.
- Structured algorithms can be described by procedures and functions.

For more details on VHDL see e.g. [7,8].

3. EA Implementation in VHDL

For all EA components described in Section 2.1, a VHDL module has been developed. First, the overall structure is outlined. Due to page limitation, a complete description of all modules cannot be given. Instead, we briefly discuss the main features of each element and then show one module, i.e. the realization of the module `fitness`, in detail. This is justified by the fact that often the fitness evaluation is the most complex operation in EAs regarding execution time. Finally, the components of the package that in sum build the library GAME-HDL are introduced.

The overall flow of the hardware realization of the EA is as follows:



The only external inputs are the clock signal `clk` and the reset signal `rst`. The clock is needed, since we consider a synchronous digital design. The reset is used for the initialization of the elements. Of course, it is easy to extend the library to also allow external elements to be added to the population during the initialization, i.e. to use hybrid EAs, where problem specific information can be added to the EA concept. For simplicity, in this application we restrict ourselves to randomly generated elements. The circuit outputs are the best fitness value (`best_fitness`), the fitness values of all elements in the population (`fit_sigs`) and the optimized elements (`elms_out`).

As originally proposed for GAs, we make use of a binary encoding, since there is a one-to-one correspondence to the digital hardware platform. For problems originally not defined over a binary alphabet, the standard encoding techniques known from GAs can be applied.

For the evolutionary operators, crossover and mutation are defined in a classical way on binary strings. A hardware implementation is very easy, since VHDL allows data types such as arrays and control structures as e.g. for-loops. Thus, only a traversal of the strings is needed which can be carried out in linear time.

In our implementation we make use of tournament selection. Here the underlying principle is so simple that a transfer in a HDL is straightforward if the fitness value is available.

In this article it is not possible to describe the full implementation of the EA. Instead, the module for the fitness computation is shown below. (For the reader not very familiar with VHDL this also gives an impression of the code structure.)

First, standard libraries are included:

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;
USE ieee.std_logic_signed.all ;
LIBRARY WORK ;
USE WORK.function_pack.all ;
```

Then, the entity is specified:

```
ENTITY fitness IS

PORT( clk              : IN BIT ;        -- clock signal
      rst              : IN BIT ;        -- reset signal
      elements         : IN pop ;        -- population
      best_fitness     : OUT INT_32 ;
      fit_sigs         : OUT INT_32_VECTOR ;
      elms_out         : OUT pop ) ;
END fitness;
```

This gives the input/output behaviour of the module (see previous section and the figure above).

The architecture describes the internal realization of the component. We used the OneMax problem, i.e. in a bit-string the number of ones is maximized, and the procedure below shows the computation of the fitness value for a given element.

```
ARCHITECTURE structure OF fitness IS

PROCEDURE calc_fitness (elements    : pop ;
                        best        : OUT INT_32) IS
            VARIABLE counter  : INT_32;
            VARIABLE max      : INT_32;
BEGIN  -- procedure for calculation of the best fitness
      max := 0 ;
      FOR i IN 0 TO element'right LOOP
            counter := 0 ;
            FOR c IN 0 TO elements(i)’right LOOP
                  IF elements(i)(c) = '1' THEN
                        counter := counter + 1;
                  END IF ;
            END LOOP ;
            IF ( max < counter ) THEN
                  max := counter ;
            END IF ;
      END LOOP ;
      best := max ;
END calc_fitness ;

SIGNAL best : INT_32 := 0 ;

BEGIN
PROCESS (clk)     -- count bits with value 1 in
                  -- elements(i) and return all fitness
                  -- values and best fitness
            VARIABLE var_best : INT_32 ;
            VARIABLE counter : INT_32 ;
      BEGIN
            IF rst = '0' THEN -- initialize fitness values
                  best_fitness <= 0 ;
                  FOR i IN 0 TO elements’right LOOP
                        fit_sigs(i) <= 0 ;
                  END LOOP ;
            ELSIF(clk'EVENT AND clk = '1') THEN
                  calc_fitness (elements, var_best) ;
                  IF ( best < var_best ) THEN
                        best <= var_best ;
                  END IF ;
                  FOR i IN 0 TO elements’right LOOP
                      counter := 0 ;
                      FOR c IN 0 TO element(i)’right LOOP
                              IF elements(i)(c) = '1' THEN
                                   counter := counter + 1;
                              END IF ;
                      END LOOP ;
                      fit_sigs(i) <= counter ;
                  END LOOP ;
                  best_fitness <= best;
            END IF ;
      END PROCESS;
END structure ;
```

The architecture of entity `fitness` consists of two main components, i.e. a procedure that computes the fitness and a process that describes the overall flow.

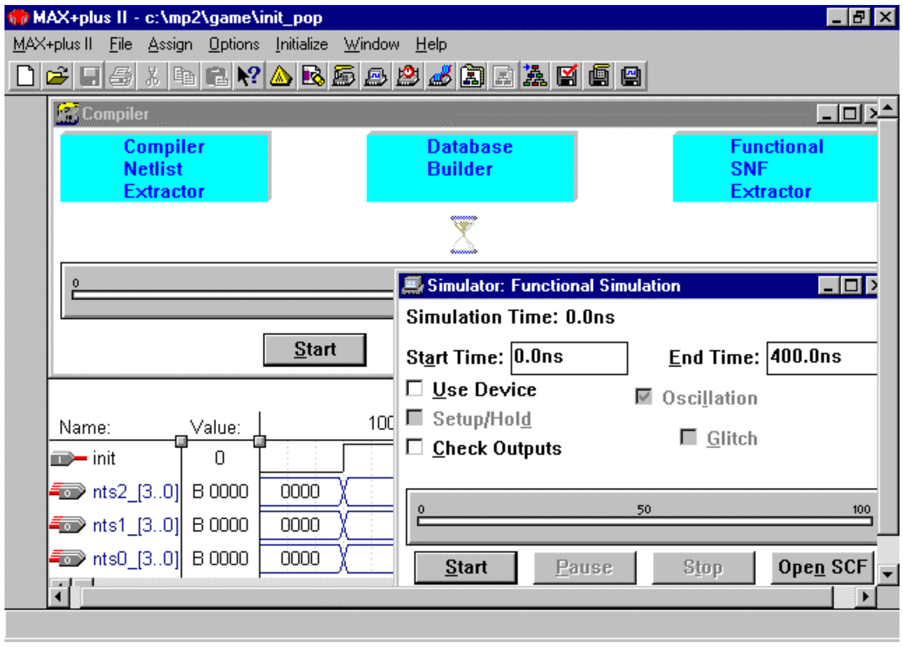
3.1. GAME-HDL: Library of Modules

In an analogous way, components for all the other modules, like selection or evolutionary operators, have been defined. They have been bound together in a reusable library. In VHDL these libraries are generated using the `PACKAGE` command. Following the ideas in [11], where a modular GA library GAME (=Genetic Algorithm Managing Environment) has been developed, the resulting package is called GAME-HDL, since the package considered here has the same underlying philosophy, i.e. a modular development of an EA environment that is motivated by the reuse of components.

4. Case Study: OneMax

The method described above has been implemented in VHDL using the tool MAX+plus II from Altera Corporation under Windows. This system has been chosen since it is available as public domain. Of course, the written code can be directly transferred to other VHDL systems, since only code was written that respects the VHDL standard.

In the following, some screenshots are given from the development environment and from simulation runs. For the initial experiments we considered the OneMax problem, that has been investigated in detail in the EA community. The goal is to maximize the number of ones in a given bit string.



Simulation Time: 0.0ns

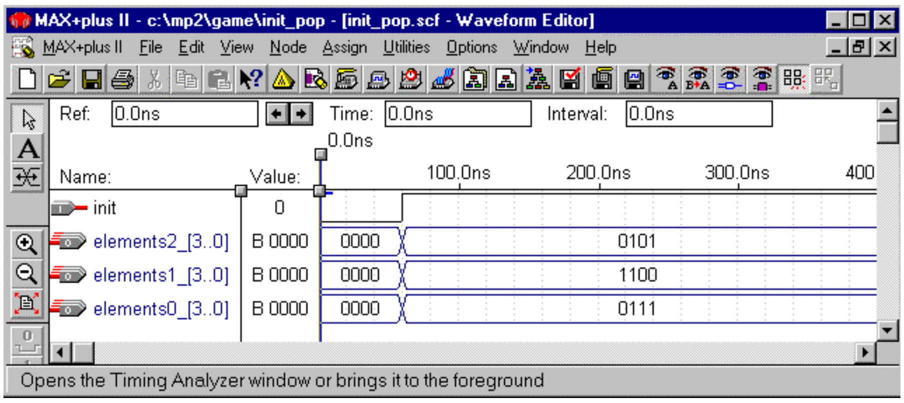
Start Time: 0.0ns
End Time: 400.0ns

☐ Use Device
☐ Setup/Hold
☐ Check Outputs
☑ Oscillation
☐ Glitch

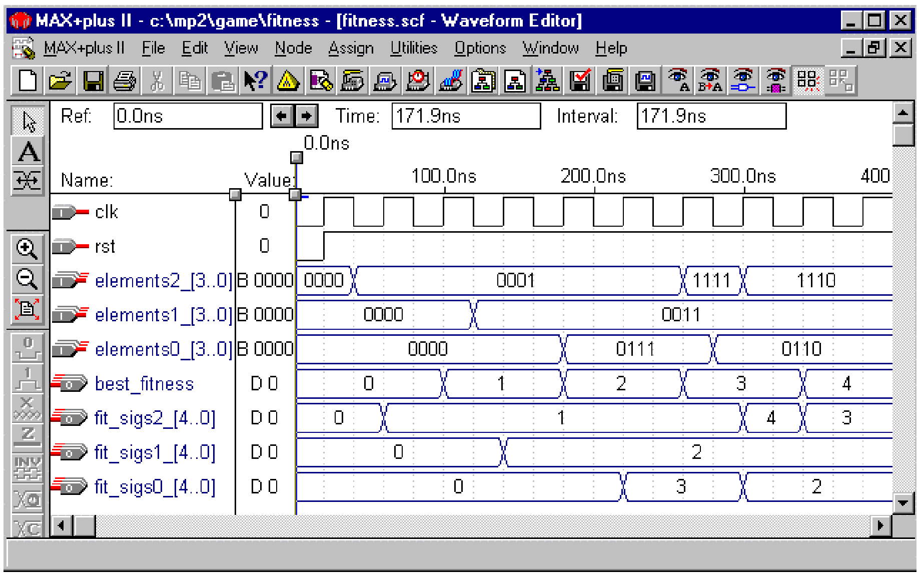
Name:	Value:	
init	0	
nts2_[3..0]	B 0000	0000
nts1_[3..0]	B 0000	0000
nts0_[3..0]	B 0000	0000

The desktop is shown above. A context -sensitive editor highlighting the keywords of the language simplifies the development. The code can be compiled and simulated in the same environment.

After the reset, the population has to be initialized. This is done by the module `init_pop` where all memory elements that are originally set to “all 0” get their values. The corresponding simulation run is shown below.


MAX+plus II - c:\mp2\game\init_pop - [init_pop.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
Ref: 0.0ns
Time: 0.0ns
Interval: 0.0ns
0.0ns
Name:
Value:
100.0ns
200.0ns
300.0ns
400
init 0
elements2_[3..0] B 0000 0000 0101
elements1_[3..0] B 0000 0000 1100
elements0_[3..0] B 0000 0000 0111
Opens the Timing Analyzer window or brings it to the foreground

The results of a simulation run for the fitness module described in Section 3 is given in the next figure:


MAX+plus II - c:\mp2\game\fitness - [fitness.scf - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
Ref: 0.0ns
Time: 171.9ns
Interval: 171.9ns
0.0ns
Name:
Value:
100.0ns
200.0ns
300.0ns
400
clk 0
rst 0
elements2_[3..0] B 0000 0000 0001 1111 1110
elements1_[3..0] B 0000 0000 0011
elements0_[3..0] B 0000 0000 0111 0110
best_fitness D 0 0 1 2 3 4
fit_sigs2_[4..0] D 0 0 1 4 3
fit_sigs1_[4..0] D 0 0 2
fit_sigs0_[4..0] D 0 0 3 2

As can be seen on the time scale on the top of the window, the execution times of the fitness evaluation can be carried out within one clock cycle (40ns). This shows the efficiency of the approach. Even better results can easily be obtained by switching to a more efficient technology by exchanging the technology dependent components in the library. Notice that no changes in the VHDL code are required. The signal `best_fitness` gives the best fitness obtained so far. The fitness function is evaluated with each rising clock edge (see above). The fitness value for `best_fitness` is available one clock cycle later.

5. Conclusions

In this paper we presented a hardware implementation of Evolutionary Algorithms based on Hardware Description Languages. A tool called GAME-HDL has been implemented in VHDL, one of the most frequently used description languages. GAME-HDL is a library of VHDL modules that can be easily integrated and reused to design hardware-based EAs.

A case study for the OneMax problem has been demonstrated. Simulation runs have been provided that gave an impression of the high execution speed of the resulting circuits.

It is focus of current work to apply GAME-HDL to more complex optimization problems and in this context also experiment with further evolutionary operators and selection principles.

References

1. R. Drechsler, Evolutionary Algorithms for VLSI CAD, Kluwer Academic Publisher, 1998

2. P. Mazumder and E. Rudnick, Genetic Algorithms for VLSI Design, Layout & Test Automation, Prentice Hall, 1998

3. M. Erba, R. Rossi, V. Liberali and A. Tettamanzi, An Evolutionary Approach to Automatic Generation of VHDL Code for Low-Power Digital Filters, EuroGP, LNCS 2038, pp. 36-50, 2001

4. R. Drechsler and N. Drechsler, Evolutionary Algorithms for Embedded System Design, Kluwer Academic Publisher, 2002

5. E. Cantu-Paz, Efficient and Accurate Parallel Genetic Algorithms, Kluwer Academic Publisher, 2000

6. S. Koizumi, S. Wakabayashi, T. Koide, K. Fujiwara, N. Imura, A RISC Processor for High-Speed Execution of Genetic Algorithms, GECCO, pp. 1338-1345, 2001

7. R. Lipsett, C.F. Schaefer and C. Ussery, VHDL: Hardware Description and Design, Kluwer Academic Publishers, 1989

8. D. Perry, VHDL, McGraw Hill, 1998

9. M. Keating and P. Bricaud, Reuse Methodology Manual for System-on-a-Chip Designs, Kluwer Academic Publishers, 1999

10. H. Shibata and N. Fujii, Analog Circuit Synthesis Based on Reuse of Topological Features of Prototype Circuits, GECCO, pp. 1205-1212, 2001

11. N. Göckel, R. Drechsler and B. Becker, GAME: A Software Environment for Using Genetic Algorithms in Circuit Design, Applications of Computer Systems, 240-247, 1997

12. J.H. Holland, Adaption in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, MI, 1975,

13. I. Rechenberg, Evolutionsstrategie, Frommann-Holzboog, 1973

14. T. Bäck, Evolutionary Algorithms in Theory and Practice, Oxford University Press, 1996

15. J. Koza, Genetic Programming - On the Programming of Computers by means of Natural Selection, MIT Press, 1992

16. L. Davis, Handbook of Genetic Algorithms, van Nostrand Reinhold, New York, 1991