Biologically-Inspired: A Rule-Based Self-Reconfiguration of a Virtex Chip

Gunnar Tufte and Pauline C. Haddow

The Norwegian University of Science and Technology
Department of Computer and Information Science
Sem Selandsvei 7-9, 7491 Trondheim, Norway
{gunnart,pauline}@idi.ntnu.no

Abstract. To be able to evolve digital circuits with complex structure and/or complex functionality we propose an artificial development process as the genotype-phenotype mapping. To realistically evolve such circuits a hardware implementation of the development process together with high-speed reconfiguration logic for phenotype implementation is presented. The hardware implementation of the development process is a programmable reconfiguration processor. The high-speed reconfiguration logic for evaluation of the phenotype is capable of exploiting the advantage of massive parallel processing due to the cellular automata like structure.

1 Introduction

In Evolvable Hardware (EHW), evolutionary algorithms can be used to evolve electronic circuits. In general, a one-to-one mapping for the genotype-phenotype transition is assumed.

Introducing inspiration from biological development, where a genotype represents a building plan of how to assemble an organism rather than a blueprint of the assembled phenotype may be a way of artificially developing complex phenotypes from relatively simple genotypes [1,2]. Combining artificial development with evolution thus aims to improve the complexity of electronic circuits that may be evolved [3,4,5].

In this work we continue to use the knowledge-rich development for circuit design on a Virtual Sblock FPGA platform [5]. The main goal herein is to define a hardware solution combining artificial development with intrinsic evolution to be able to realistic evolve complex structure and/or complex functionality.

The hardware used in this project is a cPCI host computer and a Nallatech BenERA [6] cPCI FPGA card featuring a Xilinx XCV1000E-6 [7].

The article is laid out as follows: Section 2 introduces the Sblock and the Virtual FPGA concept, which is the platform for our development process. Section 3 describes the mapping of Sblocks to our target technology. Section 4 describes our development process. Section 5 explains the implementation of our hardware solution combining the development process and the developed Sblock phenotype on an FPGA. Finaly a conclusion is presented in Section 6.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 1249–1256, 2004.

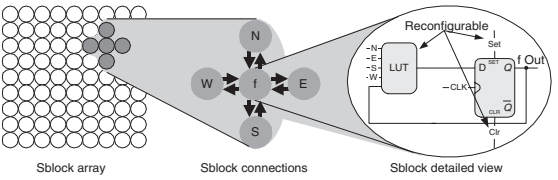

Fig. 1. Sblocks in a Virtual Sblock FPGA

2 Virtual Sblock FPGA

The Virtual Sblock FPGA is a technology independent platform for evolvable hardware. The key feature is a more evolution and development friendly hardware platform for evolving digital circuits than what is offered from commercial FPGAs. Figure 1 illustrates the Virtual Sblock FPGA: the grid of Sblocks; local connections between neighbouring Sblocks and the internal logic of an Sblock.

The Virtual Sblock FPGA [8] contains blocks — *Sblocks* — laid out as a symmetric grid where neighbouring blocks are connected. Each Sblock neighbours onto Sblocks on its four sides (north N, east E, south S and west W). The output value of an Sblock is synchronously updated and sent to all its four neighbours and as a feedback to itself.

An Sblock may be configured as either a logic or memory element with direct connections to its four neighbours or it may be configured as a routing element. Functionality of an Sblock (Sblock type) is defined by the content of its look-up table (LUT), a function generator capable of generating all possible five-input boolean functions. The five inputs to the LUT consist of inputs from the 4 neighbours and its own output value (Sblock state).

The Virtual Sblock FPGA may also be viewed at as a two-dimensional cellular automata (CA). The cellular automata may be uniform, if all Sblocks have the same LUT configuration, or non-uniform if different Sblocks have different LUT configurations.

In this work, the cellular automata interpretation is chosen. The CA i.e. the complete circuit, starts from some initial condition (interpreted as the input) and runs for a number of interactions to some final condition (interpreted as the output). This is the most common interpretation of CA computation [9]. This interpretation is used in [10,5].

A detailed description of Sblocks and the Virtual Sblock FPGA is presented in [8]. Evolution and development friendly features of the platform may be found in [11]. The term development friendly is used to reflect the fact that properties of the architecture are more suitable to developmental techniques than those found in today's technology.

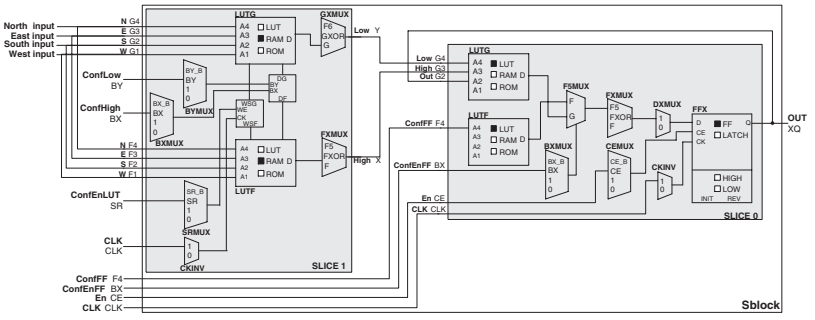

Fig. 2. Sblocks hard macro implementation for on-chip reconfiguration

3 Sblock Architecture on a Virtex-E FPGA

The mapping of the Sblock architecture to an FPGA aims to utilize the underlying technology of the target device. The Virtex-E [7] series of FPGAs from Xilinx consists of an array of configurable logic blocks (CLBs) and fast internal BlockRam (BRAM) memory. BRAM modules are placed in-between CLB columns at fixed intervals. Exploiting the 2-d array of CLBs in the device, we form the 2-d grid of Sblocks where an Sblock fits into one CLB. This adaptation gives us the possibility to constrain the placement of Sblocks to specific CLB locations in the target device. Knowledge of placement gives us two advantages. First, knowledge of Sblock placement is crucial for partial reconfiguration, since it is essential to know where the desired Sblock is located to be able to update the LUT. Second, the constrained placement of Sblocks ensures adjacent Sblocks to be placed in CLBs according to their neighbourhood. This implies short and conform routing between Sblocks, giving approximately the same propagation delay for all Sblock connections. The Sblock implementation is shown in Figure 2.

Configuration of an Sblock FPGA is a two stage process. The first stage configures fixed logic, assigns and configures Sblocks into their respective CLBs, and sets up the routing resources to connect Sblocks. In the second stage reconfiguration manipulates LUTs in desired Sblocks, utilizing partial dynamic reconfiguration of Sblocks. The first configuration stage uses standard configuration mode provided by the device. The second configuration stage may use the standard partial reconfiguration modes i.e. frame based reconfiguration [12], or custom reconfiguration as explained in Section 5.

The LUTs are used as both function generators and as shift registers. An Sblock requires two slices, while one Sblock fits in to one CLB using both slices. Bold letters indicate input and output ports on the slices. The LUTs inputs are connected to form the five input LUT, due to the use of the *BXMUX* in the shift register implementation the five input LUT here consists of $LUTG$ and $LUTF$ in slice 1 and $LUTG$ in slice 0. The shown Sblock implementation gives a maximum number of Sblocks on an FPGA that equals the number of CLBs.

Extra signals in Figure 2 are data and control signals for the custom reconfiguration of the Sblocks. Instead of using frames to alter LUT content and flip-flop state we have added extra logic on the FPGA to shift configuration data in and out of the Sblocks. The data port for shifting data into the LUTs is split in two ports: *ConfLow* for shifting data into *LUTG* and *ConfHigh* for shifting data into *LUTF*. *ConfEnLut* enables reconfiguration of LUTs. The *ConfFF* is a data input used to shift in states to the flip-flop when *ConfEnFF* is active.

The function of the Sblock in Figure 2 can of course be reconfigured by changing LUT bits by reading and writing frames, but the implementation also supports a custom reconfiguration mode for high-speed reconfiguration. The custom reconfiguration mode exploits the possibility of using LUTs as both function generators and shift registers. Instead of altering LUT contents by writing frames, on-chip reconfiguration logic can address desired Sblocks and alter their LUTs by shifting in new data.

4 Development on an Sblock Platform Inspired by Biology

As stated in Section 1, instead of using a genotype representation as a blueprint for a circuit a development process uses a building plan of how to assemble the circuits. In this work the building plan consists of rules.

The rules are based on two types of rules i.e. change and growth rules. These rules are restricted to expressions consisting of the type of the target cell and the types of the cells in its neighbourhood. Firing of a rule can cause the target cell to change type, die (implemented as a change of type) or cause another cell to grow into it.

Figure 3 illustrates the process of applying growth and change rules. A growth rule targets the **C** Sblock which is an empty Sblock. This means that the condition of this change rule, including types of the four neighbours and the target itself match. The rule triggered grows a new Sblock into *C* by copying the Sblock type from the *E* neighbour. A change rule then targets the Sblock in position *C*. The change rule result is that the targeted Sblock *C* is changed.

The direction of growth is built into the rules themselves [5]. Growth is not triggered by an internal cell in the growing organism but in fact by a free cell — empty Sblock, neighbouring the growing organism. When a growth rule triggers, it matches to a free cell and its given neighbourhood. Four different growth rules

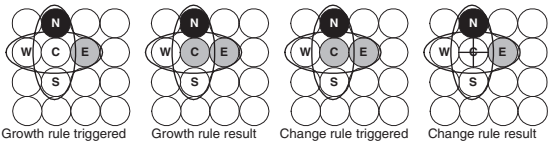

Fig. 3. Change and Growth on an Sblock Grid

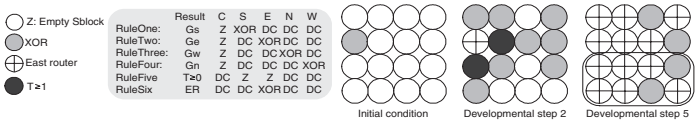

Fig. 4. Example of development to a symmetrical organism

control growth from the North (G_N), East (G_E), South (G_S) and West (G_W) respectively. This means that if, for example, a north growth rule targets a free cell, then the type of the cell to the North is copied to the free cell.

In Figure 4 development of an organism is shown. The example shows development of an organism with structural properties, no functionality is considered. Three different Sblock types are possible: XOR, East Router (ER), and a threshold element ($T \geq 1$).

The organism develops from a single cell (axiom) shown in the initial condition. After two development steps, the organism has grown from the axiom to eleven Sblocks and differentiated to consist of all three Sblock types. At the fifth and final development step, the organism has developed into an organism with two symmetrical patterns (one of the two patterns is indicated). The genome (rules) used is as shown. The rules are ranked, rule one has the highest priority and the last rule lowest priority. Details concerning development of structural organisms can be found in [13] and for functionality in [5].

5 Rule Based Reconfiguration Inspired by Biology

The reconfiguration speed depends on both the time spent on downloading configuration data and on the time spent to do the reconfiguration by changing the functionality of the device. Inspired by the building plan idea we propose a small on-chip processor [14] for handling the reconfiguration. The FPGA is partitioned in two: a reconfigurable and a fixed logic partition. The Processor is the fixed partition. The reconfigurable partition is an Sblock array.

Following the idea of using a building plan as a genome the only configuration data needed may be the instructions of how to build the phenotype i.e. the rules. The on-chip reconfiguration logic is designed to use the rules to develop a phenotype in the on-chip Sblock array. The functionality of the phenotype may be the interpretation of the states of the Sblocks in the developing organism [5].

The use of a processor design as reconfiguration logic has several advantages. The two most important features may be: First the speed of downloading reconfiguration data. The compact rule representation requires only a small amount of data to be transferred to the chip. Second, the processor design allows an instruction set. Using an instruction set is flexible, making it possible to use the processor to configure Sblocks by addressing target Sblocks and downloading Sblock types to the chip. A more powerful method is to use the processor to configure the Sblock array using developmental rules. Another flexibility is the possibility to write small programs. Such programs may be setting a number of

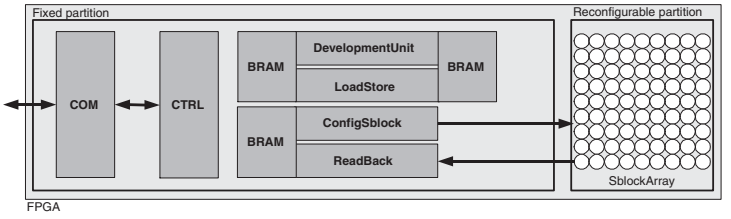

Fig. 5. Implementation of Sblock on-chip reconfiguration

development steps and/or clock cycles to run the Sblock array. Programs can also be used to monitor the Sblock array during the development process.

The possibility a hardware implementation gives for massive parallel processing in CA like structures is exploited for running the Sblock array, the development process where parallel processing includes implementing the rules in every cell, was considered to resource consuming. Therefore the development process is sequential.

Figure 5 shows a block drawing of our reconfiguration processor together with the Sblock array. The Sblocks is implemented as shown in Figure 2, where reconfiguration of the Sblocks is done by shifting data into the LUTs. Running state steps is done in parallel i.e. all Sblocks output values are updated in parallel each cycle. The processor is divided into seven modules:

- COM: Interface for external communication with the host on the cPCI bus.
- CTRL: This is the control unit of the processor. This module also includes the instruction memory and manages communication with the *COM* module.
- BRAM: Intermediate storage of type and state of the Sblock array. Split in two BRAM modules, storing current development steps state and type and a second for the developing organism.
- DevelopmentUnit: Hardware implementation of the development process in Section 4. To speedup the process the development process works on a copy of the actual Sblock array and the copies is stored in BRAM. The current Sblock array configuration is stored in one *BRAM* module. The development process reads the current Sblock array and updates the second *BRAM* module with new Sblock types for those that triggers a rule. The *DevelopmentUnit* is a six stage pipeline, processing two Sblocks each clock cycle.
- ConfigSblock: Reconfiguration of the Sblock array. The module reads out Sblock types and states from *BRAM* and reconfigures the Sblocks with the Sblock array under development stored in *BRAM*. Implemented as a seven stage pipeline, reconfigures two Sblocks every cycle.
- LoadStore: Handles reading and writing of Sblock types and states to *BRAM* and communicates with external devices through the *COM* interface.
- ReadBack: Used to monitor Sblock types and states. Read back of Sblock states is done from the Sblock array by the *Readback* module and stored as state information in *BRAM*. Sblock types are already present in *BRAM* from the latest development step.

A typical experiment for the system will be to run a genetic algorithm (GA) on the host computer and the development process with the developing phenotype in hardware. As stated in Section 3 the configuration process is a two stage process, where the first stage configures the FPGA with the processor design and the Sblock array. The configuration in the first stage is done after synthesis of the processor design with an Sblock array of desired size. After the first configuration stage, the FPGA contains the reconfiguration processor and the Sblock array.

Before the experiments starts, the development process must be defined i.e. defining the number of development steps and number of state steps for each evaluation. Criteria for read back of information from the evaluation can also be defined. These definitions are done by writing and downloading a program to the reconfiguration processor. The downloaded program is executed to evaluate each downloaded rule genome from the GA in the host.

To evaluate our hardware solution we first compared reconfiguration time using the configuration processor with Virtex frame based reconfiguration using the JTAG interface. The task was to reconfigure a 32 by 32 Sblock array. Since the goal is to evaluate configuration speed we wanted to include both download and reconfiguration time in both cases. To manage this the reconfiguration processor was only used to receive configuration data from the host and configure the addressed Sblock i.e. all reconfiguration data was in full downloaded from the host for both cases. The test was done by reconfiguring all Sblocks in the array a 1000 times. The frame based reconfiguration used 170,38 seconds to accomplish the task. The configuration processor accomplished the same task in 0.55 seconds, giving a speedup of more then 300.

A comparison of performance of the system with a software simulation was also tried. A software simulator simulated both the development process and the Sblock states on an 8 by 8 Sblock array. The experiment included both development steps and state steps. The development process was 10 000 steps with 50 000 state steps at each development step. For each development step all Sblock types in the array was stored together with the last state step information. Storing information implies writing back data from the FPGA to a file on the host over the cPCI bus while for the software simulation storing is writing to a file. The software simulation accomplished the task in 18.06 minutes, the runtime in hardware was measured to 6.2 seconds giving a speedup of 175 times. The software simulation was performed on a 2.3 GHz Pentium 4 PC.

6 Conclusion

The presented hardware system consisting of a hardware platform including a development process and high-speed reconfigurable logic shows a promising speedup both for intrinsic evolution and for evolution using development as genotype-phenotype mapping. The implementation of a reconfiguration processor is capable of both running intrinsic EHW experiments requiring high-speed reconfiguration and to speedup the genotype-phenotype mapping.

References

1. Kitano, H.: Building complex systems using development process: An engineering approach. In: Evolvable Systems: from Biology to Hardware, ICES. (1998) 218–229
2. Bentley, P.J., Kumar, S.: Three ways to grow designs: A comparison of embryogenies for an evolutionary design problem. In: Genetic and Evolutionary Computation Conference (GECCO '99). (1999) 35–43
3. Gordon, T.G.W., Bentley, P.J.: Towards development in evolvable hardware. In: the 2002 NASA/DOD Conference on Evolvable Hardware (EH'02). (2002) 241 –250
4. Miller, J.F., Thomson, P.: A developmental method for growing graphs and circuits. In: 5th International Conference on Evolvable Systems (ICES03). (2003) 93–104
5. Tufte, G., Haddow, P.C.: Identification of functionality during development on a virtual sblock fpga. In: Congress on Evolutionary Computation. (2003)
6. Nallatech: BenERA User Guide. Nt107-0072 (issue 3) 09-04-2002 edn. (2002)
7. Xilinx: Xilinx Virtex-E 1.8 V Field Programmable Gate Arrays Production Product Specification. Ds022-1 (v2.3) july 17, 2002 edn. (2002)
8. Haddow, P.C., Tufte, G.: Bridging the genotype-phenotype mapping for digital FPGAs. In: the 3rd NASA/DoD Workshop on Evolvable Hardware. (2001) 109–115
9. Mitchell, M., Hraber, P.T., Crutchfield, J.P.: revisiting the egde of chaos: Evolving cellular automata to perform computations. Complex Systems **7** (1993) 89–130 Santa Fe Institute Working Paper 93-03-014.
10. van Remortel, P., Lenaerts, T., Manderick, B.: Lineage and induction in the Development of evolved genotypes for non-uniform 2d cas. In: 15th Australian Joint Conference on Artificial Intelligence. (2002) 321–332
11. Haddow, P.C., Tufte, G., van Remortel, P.: Evolvable hardware: pumping life into dead silicon. In Kumar, S., ed.: On Growth, Form and Computers. Elsevier Limited Oxford UK (2003) 404–422
12. Xilinx: Xilinx XAPP 151 Virtex Configuration Architecture Advanced Users' Guide. 1.1 edn. (1999)
13. Tufte, G., Haddow, P.C.: Building knowledg into developmental rules for circuite design. In: 5th International Conference on Evolvable Systems ICES. (2003) 69–80
14. Djupdal, A.: Design and Implementation of Hardware Suitable for Sblock Based Experiments, Masters Thesis. The University of Science and technology, Norway (2003)