Speeding up Hardware Evolution: A Coprocessor for Evolutionary Algorithms

Tillmann Schmitz, Steffen Hohmann, Karlheinz Meier, Johannes Schemmel, and Felix Schürmann

University of Heidelberg, Kirchhoff Institute for Physics,
INF 227, D-69120 Heidelberg
tschmitz@ix.urz.uni-heidelberg.de
http://www.kip.uni-heidelberg.de/vision

Abstract. This paper proposes a coprocessor architecture to speed up hardware evolution. It is designed to be implemented in an FPGA with an integrated microprocessor core. The coprocessor resides in the configurable logic, it can execute common genetic operators like crossover and mutation with a targeted data throughput of 420 MByte/s. Together with the microprocessor core, a complex evolutionary algorithm can be developed in software, but is processed at the speed of dedicated hardware.

1 Introduction

An evolvable hardware platform usually consists of three main parts: the reconfigurable hardware (RH) and two processing units: the fitness calculation unit (FCU) and the evolutionary algorithm unit (EAU). There are different possibilities to implement the processing units (FCU and EAU) like PC-based approaches ([1], [2]), DSPs [3], FPGAs [4] and ASICs.

These implementations can be classified whether they are software- (PC and DSP) or hardware-oriented (FPGA and ASIC). While the first option provides the advantage of variability, easier design and maintainability, it is relatively slow. A hardware implementation allows a higher degree of parallelism and is therefore faster, but usually needs more design effort.

Our group uses an analog neural network ASIC (ANN) [5] as the reconfigurable architecture (see Fig. 1). Its configuration stream consists of about 45 kByte of data and it calculates one network layer every 20 ns. To exploit the network's speed, substantial computing power and data-flow rate are needed to generate new generations in time.

In this paper we propose a coprocessor architecture to bring up the evolutionary algorithm to the speed of dedicated hardware. It is customized to suit the demands of our ANN, but the architecture is flexible enough to be transferred to other reconfigurable hardware systems. The coprocessor is designed for a Virtex-II Pro FPGA manufactured by Xilinx [6]. This FPGA provides programmable logic cells, high speed serial links for off-chip communication and an

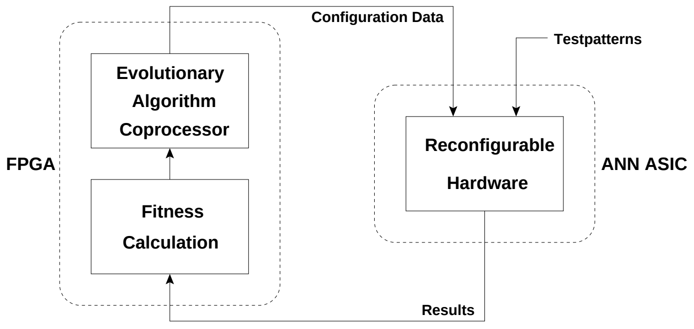

Fig. 1. Setup used for the evolvable hardware experiments

IBM PowerPC 405 core immersed in the FPGA fabric. With this device we are able to partition between soft- and hardware:

The evolutionary algorithm is written in software and executed in the microprocessor core. The coprocessor resides in the programmable logic cells and is coded in a hardware description language. The problem-specific fitness calculation is not a topic in this paper. It is done either by software in the microprocessor core or in specialized hardware in the FPGA.

The instruction set and the internal structure of the coprocessor are designed to allow a wide range of evolutionary operators. This makes it possible to use the same hardware configuration code for a large number of different experiments and training algorithms.

With this coprocessor system it is possible to design complex evolutionary algorithms in a software environment, but let the time-expensive parts be executed in hardware. This yields a data throughput of up to 420 MByte/s of genetic data.

2 Evolution System

Our hardware integrates a Xilinx FPGA Virtex-II Pro [6] together with 256 MByte of DDR-SDRAM [7] and an analog neural network ASIC on a single PCB[1], as shown in Fig. 2. It is interfaced to a host computer. This connection is used only to initialize the microprocessor core, the random generators, etc. and to monitor the evolution progress.

Analog Neural Network ASIC. The target of the evolution is a neural network. It is implemented in a mixed-signal ASIC developed by our group, partitioned in four network blocks containing 128 input neurons and 64 output neurons each. This sums up to 32768 synapses. Taking into account the resolution of 11 bit per synapse, our configuration bit stream, i.e. the genome, is 360448 bit (44 kByte) long (see [5]).

[1] Printed Circuit Board (PCB)

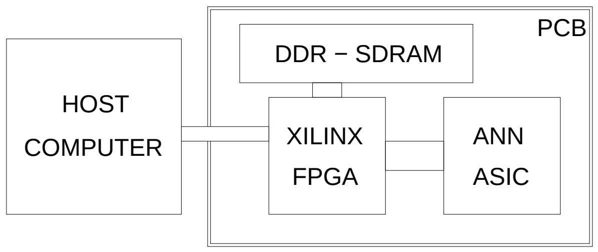

Fig. 2. Evolution system

This ASIC is connected to the FPGA by a parallel LVDS[2] link, capable of transferring up to 1.2 GByte/s of configuration or testpattern data. To evaluate an individual or genome, the configuration data has to be transferred to the ASIC first. After this is done, the testpatterns are applied successively and finally the results for each testpattern are sent back to the FPGA across the LVDS link.

Xilinx FPGA Virtex-II Pro. The coprocessor is designed for the Virtex-II Pro FPGA. Apart from the programmable logic, it contains 504 kBit internal dual-ported SRAM, hardware multiplier and high speed serial links for off-chip communication. Its outstanding feature is the PowerPC, a microprocessor core delivering 420 Dhrystone MIPS. As the PowerPC is immersed in the FPGA fabric, it can access part of the internal SRAM as fast as its own cache. Our coprocessor connects to the additional SRAM port, i.e. we use the dual ported internal SRAM to communicate between microprocessor core and coprocessor.

Peripherals and Scalability. The internal SRAM provided by the FPGA is neither sufficient to store the genome data nor the testpatterns. Therefore, we added a DDR-SDRAM module to the PCB for the FPGA to use up to 1 GByte of external memory at a transfer rate of up to 2.1 GByte/s.

The evolution system is designed in a way that allows to combine 16 systems on one backplane. These modules are connected to each other with the serial links described above. Each FPGA offers four links, we are targeting a 16 node, 2-dim torus, with each connection transferring up to 3.125 Gbit/s.

3 Evolutionary Algorithm Coprocessor (EAC)

3.1 Overview

The EAC pursues two aims: It must process a high data throughput and it has to be parameterized to allow a wide range of evolutionary algorithms without changing the hardware configuration. The first aim demands a pipelined structure. To satisfy the second this pipeline is controlled and managed by a set of

[2] Low-Voltage Differential Signalling (LVDS)

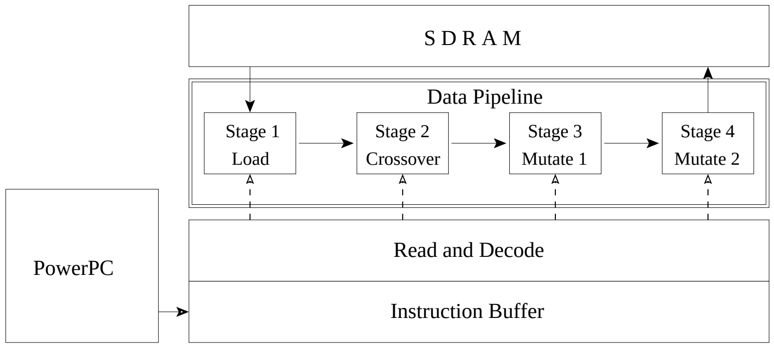

Fig. 3. Instruction buffer and command structure

instructions. This structure is shown in Fig. 3. The microprocessor core addresses the EAC by writing instructions to the *Instruction Buffer* (IB). They are read and decoded successively. Depending on the instruction they control different parts of the pipeline, indicated with dashed arrows in Fig. 3.

3.2 Data Flow Control

All genetic data is stored in the SDRAM. A population consists of a number of individuals, represented by their *genomes*. Each genome is divided into *chromosomes*. A chromosome is stored consecutively in the SDRAM and crossover takes place inside one chromosome. Each chromosome contains an arbitrary number of genes, the smallest unit of the genome. The mutation operator works on single genes. A typical ANN training setup uses 10-100 individuals and one chromosome per neuron. The coprocessor can handle any combination that fits into memory.

A more detailed view of the pipeline and the surrounding control structures is given in Fig. 4. The genes are read from the SDRAM as parents, modified in one crossover and two mutation stages and written back as children to a new address in the SDRAM.

In order to feed genetic data into the pipeline (or to write it back to the memory), the software must specify the SDRAM address and the number of genes to be transferred. The SDRAM works burst-oriented, i.e. consecutive addresses can be accessed very fast while random access is relatively slow. Therefore, the chromosomes are stored consecutively in the SDRAM, and FIFOs accumulate the data in order to address the SDRAM in bursts.

Parallelism. A gene is expressed with 11 bit resolution. As described below, we need two additional bits to control the evolutionary operators. To simplify the addressing, each gene occupies two bytes, so three bits are available for future use.

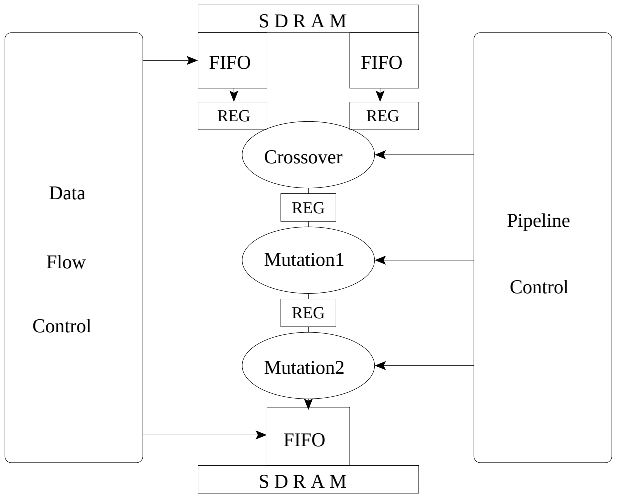

Fig. 4. The pipeline with its control structure

The design is targeted for 80 MHz. Four pipelines work in parallel to take advantage of the SDRAM transfer rate. Fig. 4 depicts one pipeline processing one gene per cycle. Provided, enough genetic data and instructions are supplied, the four pipelines can process four genes per cycle.

3.3 Combinatorial Logic

Fig. 5 shows a detailed description of the evolution pipeline. There are two different kinds of control options:

- *Multiplexer Control Bits*
- *Global Parameters*

The *Multiplexer Control Bits* (`SelectParent`, `SelectRandom`, `SelectScale`, `SelectConstant`, `SelectReplace` and `SelectMutate`) in Fig. 5 are numbered from one to six, they may change with every new gene. For example, `SelectMutate` decides whether the original gene is passed down unaltered or undergoes mutation. As this is done for each gene independently, `SelectMutate` changes almost with every new gene.

The *Global Parameters* (`RandomMaximum`, `Constant` and `Fraction`) are recognizable by dashed boxes. They are at least valid for a whole chromosome, in most cases for the entire evolution. `RandomMaximum` for example may be set to 1023 ($2^{10} - 1$) at the beginning to get mutated genes with every possible value. Later, as the fitness increases, `RandomMaximum` might be decreased to fine-tune the population.

This difference in the update frequency is important. The *Global Parameters* are updated infrequently, therefore, the software can set them with instructions.

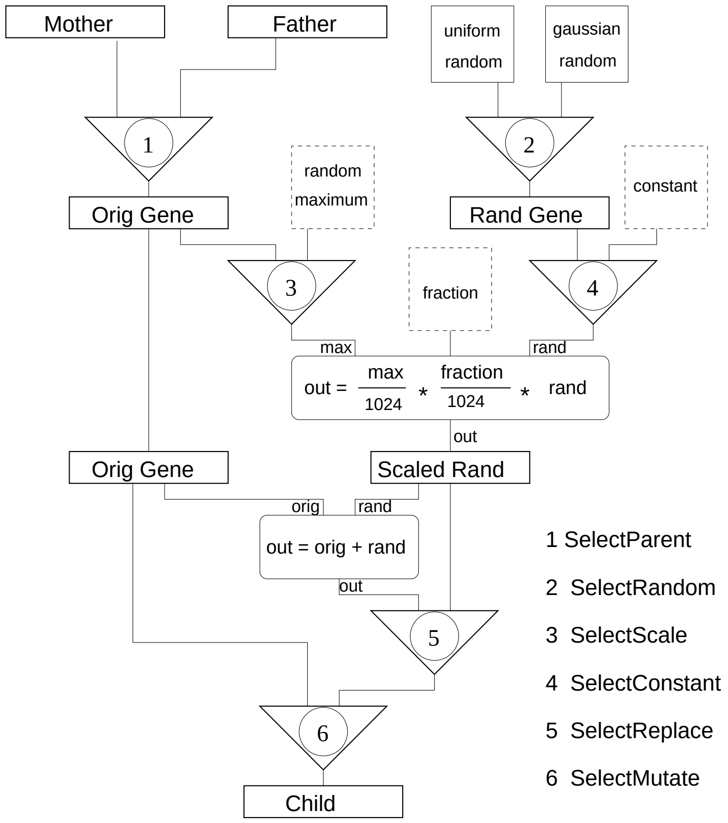

Fig. 5. The pipeline in detail

Since the *Multiplexer Control Bits* may be updated every cycle, they must be generated faster to achieve a substantial data throughput.

Multiplexer Control Bits (MCB). With the setting of the MCBs, one can

1. choose from which parent the child inherits the gene (`SelectParent`).
2. choose between a uniform or gaussian random distribution (`SelectRandom`).
3. scale the new gene value (`SelectScale`).
4. choose between a random or a constant new gene (`SelectConstant`).
5. choose whether the generated gene shall be added to the original or replace it (`SelectReplace`).
6. decide whether a mutation occurs or not (`SelectMutate`).

As stated above, the MCBs may change their values with every new gene, i.e. every new cycle. Thus they cannot be set individually by software instructions since this would be too slow. On the other hand, the MCBs actually define the

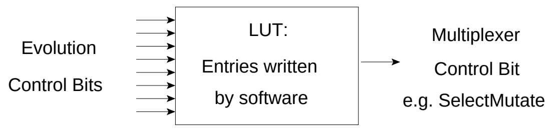

Fig. 6. A LUT sets the MCBs

underlying evolutionary algorithm. Still the description of the algorithm must be based in software to be variable.

This contradiction is solved by introducing an additional set of bits, the *Evolution Control Bits* (ECB). These ECBs generate the MCBs via a look-up table (LUT): The LUT address is given by the ECBs, the LUT entries are written by software, and the LUT outputs equals the MCBs (Fig. 6). There are three kinds of ECBs:

- Instruction dependent bits
  Two *Evolution Control Bits*, Mask1 and Mask2 are set by instructions. For example they can be used to set crossover points: To transfer the necessary data for a two-point crossover with crossings after the 27th and 78th bit for a chromosome with 128 genes, the run-length-coded mask would be (27,51,50).
- Random bits
  Two *Evolution Control Bits*, Rand1 and Rand2, are generated by a random generator each. They are used to decide whether a mutation occurs or not. Their output equals '1' with a probability given by an instruction.
- Gene dependent bits
  Four *Evolution Control Bits*, `Mother1`, `Mother2`, `Father1` and `Father2` are obtained from the parent genes. They can be used to steer the evolution, for example, to exclude a single gene from mutation if Mother1 and Father1 equals '0'. As stated above, our genes are 11 bit long plus two *Evolution Control Bits* for each gene.

Now each of the six *Multiplexer Control Bits* depends on the eight *Evolution Control Bits*. The exact dependency is controlled by the software. We are using a look-up table (LUT) based structure, where the LUT address is given by the *Evolution Control Bits*.

For each *Multiplexer Control Bit* one 8 bit LUT is needed. The ECBs form the input or address to that LUT, while the software specifies the LUT entries. Note that these entries are generally only written once, but can be changed if the necessity arises.

This is best understood with an example: A mutation algorithm shall use two different mutation rates, 2% and 5%. There are genes with the high mutation rate, genes subject to the low mutation rate and genes without any mutation at all. The Mother1 and Father1 bits are used to differentiate between the three cases. A gene shall be mutated with 5% probability if both father1 and mother1 bits are set to '1', but with a probability of 2% if either the father's bit or the mother's bit is set to '1'. No mutation shall occur if both bits are set to '0'.

The LUT implementing this behavior is shown in Tab. 1 (This example uses only a 4-bit, instead of an 8-bit LUT as described in the text. The remaining four *Evolution Control Bits* (Mother2, Father2, Mask1 and Mask2) are treated as don't cares for this setup).

Table 1. Example LUT with four LUT-Address-Bits. An 'X' in the table denotes 'don't care'. To obtain the full LUT, all rows with an 'X' must be duplicated and '0', resp. '1' has to be inserted instead of the 'X'.

Father Bit1	Mother Bit1	Rand1 5%	Rand2 2%	Do Mutate	Explanation
0	0	X	X	0	Both gene bits '0': no mutation, independent of the state of the rand bits
1	0	X	0	0	Exactly one gene bit '1':
1	0	X	1	1	mutation occurs if Rand2 (2%)
0	1	X	0	0	equals '1', i.e. with a
0	1	X	1	1	probability of 2%
1	1	0	X	0	Both gene bits '1': mutation
1	1	1	X	1	occurs if Rand1 (5%) equals '1'

3.4 Memory Organization and Instruction Set

Memory Organization. Our system offers two kinds of memory, FPGA internal SRAM (504 kBit) and off-chip DDR-SDRAM (up to 1 GByte). The first is used to communicate between processor core and Coprocessor. It is dual ported and can be randomly accessed independently from both ports. The genetic data, i.e. the genomes, are stored in the off-chip SDRAM, they are far too big to fit into the FPGA.

The IB is 32-bit wide. It holds the instructions and additional data and addresses, which are used for indirect addressing, as described below.

Instruction Set. The IB does not only hold the instructions, but can also contain data and addresses. Some instructions do not transfer all their information directly. To transfer the LUT entries, the microprocessor core first writes them into the data section of the IB. After that, it issues the instruction `SetLUTSelectParent(IB address)`. Now the coprocessor looks in the IB at the transmitted address for the data. This indirect addressing is done for two reasons:

First, some instructions must transfer more than 32 bit of data to the coprocessor. To keep a consistent instruction length of 32 bit, this data has to be transferred indirectly.

Table 2. Coprocessor instruction set

Instruction	Parameters	Representation
SetLUTSelectParent SetLUTSelectRandom SetLUTSelectScale SetLUTSelectConst SetLUTSelectReplace SetLUTSelectMutate	(IB address) (IB address) (IB address) (IB address) (IB address) (IB address)	Instruction identifier bit(0-7) Instruction Buffer Address: bit(8-18)
SetRandomMaximum SetFraction SetConstant SetOneProbability1/2	(RandomMaximum) (Fraction) (Constant) (probability1/2)	Instruction identifier bit(0-7) Value: bit(8-23)
MaskIndirectBin MaskIndirectRunLength	(IB adr, words to read) (IB adr, words to read)	Instruction identifier bit(0-7) Instruction Buffer Address: bit(8-18) words to read: bit (19-31)
MaskDirectRunLength0 MaskDirectRunLength1	(quantity of zeros) (quantity of ones)	Instruction identifier bit(0-7) quantity of ones/zeros: bit (8-23)
ReadParent1 ReadParent2 WriteChild	(IB adr) (IB adr) (IB adr)	Instruction identifier bit(0-7) Instruction Buffer Address bit(8-18)

Second, using indirect addressing enables the microprocessor core to generate an address table in the IB. Depending on how many memory locations for chromosomes the algorithm needs, SDRAM addresses are written into the IB, followed by the length (i.e. the number of genes) per chromosome. To feed a sequence of genetic data into the pipeline during the evolution, the microprocessor core only transmits the corresponding IB addresses. The coprocessor then reads the SDRAM address and the number of genes to transfer from the IB. This simplifies the address management and reduces address transmitting overhead. The instruction set can be divided into two parts (see Tab. 2):

The first group of instructions must be given once to initialize the coprocessor with valid data. The LUT entries are written with an indirect addressing scheme. The *Global Parameters* must be specified directly. Both may change infrequently during the evolution.

The second group of instructions is given once for each chromosome. A mask must be given to define the crossover point(s). A mask can be defined directly or indirectly, using a run-length-code or by transferring all mask bits in binary.

Finally, the data flow control must be told where to get the parents, where to store the child and how many genes to transfer. This is done indirectly. As mentioned above, an address table in the IB is created. All data flow control instructions refer to that table.

This instruction set provides a wide range of possibilities for the software to implement various evolutionary algorithms. For example, the initial population can be generated just by setting the mutation rate to 100%, crossover operators using 3 or even 4 parent can be implemented with two (or three) 2-parent crossover operators in succession. It is also possible to just add a fraction or a constant value to each gene.

3.5 Performance Analysis

To estimate the performance of our evolution system we must consider not only the pipeline data throughput but the SDRAM transfer-rate and the microprocessor core performance as well.

– The pipeline is able to process four 11-bit-genes per cycle. With the targeted FPGA frequency of 80 MHz this sums up to 320 million genes or 420 MByte per second.
– Each gene must be read twice and written once, leading to an SDRAM transfer rate of approximately 1.26 GByte/s. This is 59% of the maximum transfer rate provided by the DDR-SDRAM.
– A chromosome on the average needs four instructions to be fed into the pipeline[3]. Considering the microprocessor core frequency of 300 MHz, a typical chromosome length of one chromosome per neuron (176 Byte for our ANN) and the pipeline data throughput of 420 MByte/s, the microprocessor core has around 125 cycles to compute and issue the four instructions mentioned above to process a chromosome and to keep pace with the pipeline.

This rough estimation shows that our system is able to run at the calculated speed of the pipeline.

4 Example: Crossover and Simple Mutation

This section presents an example of how to instruct the coprocessor to perform a crossover followed by a simple mutation. The software sets the crossover points directly and demands a mutation rate of 5%.

Tab. 3 shows the complete LUT, which is quite simple: All four parent bits, the `MaskBit2` and the `RandomBit2` are not used. The `SelectParent` equals `MaskBit1` and the `SelectMutate` equals `RandBit1`. All other MCBs are constant, `SelectRandom` is '0' to choose a uniform distribution, `SelectScale` is '1' to scale with `RandomMaximum`, `SelectConstant` is '0' to use a randomly chosen new gene, `SelectReplace` is '1', i.e. in case of a mutation we replace the original gene, instead of adding something to it.

[3] These four instructions are: `ReadParent1`, `ReadParent2`, `WriteChild` and one mask instruction.

Table 3. Example LUT with four LUT-Address-Bits (X : don't care)

Mask Bit1	Mask Bit2	Rand Bit1	Rand Bit2	Mother Bit1/2	Father Bit1/2	Select Parent	Select Rand	Select Scale	Select Const	Select Replace	Select Mutate
0	X	0	X	X	X	0	1	0	1	0	0
1	X	0	X	X	X	1	1	0	1	0	0
0	X	1	X	X	X	0	1	0	1	0	1
1	X	1	X	X	X	1	1	0	1	0	1

Global Parameters. Our mutated genes are completely random with mutation rate 5%, they cover all 11 bit, therefore we set the instructions:

```
SetRandomMaximum (1023);
SetOneProbability1 (51);
```

Note that the remaining *Global Parameters* are not used in this setup and that the mutation probability is not expressed in per cent, but in per 1024 (51 / 1024 $\simeq$ 0.05).

Specifying the Crossover Mask. The `SelectParent` LUT is programmed to let the `MaskBit1` decide from which parent the child inherits its gene. This `MaskBit1` is set with:

```
for (ChromoCount=0; ChromoCount<MaxChromo; ChromoCount++){
    MyRand = Random(128); // between [1..128]
    MaskDirectRunLength0(MyRand);
    MaskDirectRunLength1(128-MyRand);}
```

Each loop cycle will take (`MyRand`) genes from the first parent and (`128-MyRand`) genes from the second.

Data Flow Commands. Prior to issuing any data flow commands, the software has to write an address table into the IB. With this table, all chromosomes are addressable by pointers. To cross, for example, the fittest with the second-fittest genome, we need to issue the instructions:

```
ReadParent1(IB_adr = Genome[Best]);
ReadParent2(IB_adr = Genome[Second]);
WriteChild (IB_adr = Genome[Unused1]);
```

Note that the pointers `Best`, `Second`, etc. in general change their value with every new generation.

A complete evolution starts with the creation of an initial population (by setting the mutation rate to 100%). This population is evaluated and ranked according to the fitness. Based on this ranking, the software decides which genomes are crossed and issues the appropriate data flow and mask instructions, as shown in the examples above. After all genomes have been crossed and mutated, the evaluation starts again. This circle is continued until a satisfactory fitness is reached.

5 Conclusion and Outlook

We have introduced a coprocessor architecture to speed up evolutionary algorithms designed for the FPGA Virtex-II Pro. The coprocessor is able to perform common genetic operators like crossover and mutation with a data throughput of up to 420 MByte/s due to pipelining and parallelism. It still has the flexibility of software. Therefore, wide range of evolutionary algorithms can be designed and maintained in software, but are processed at high speed in dedicated hardware.

In the future we plan to implement additional operators, e.g. the possibility to add two genes together during the crossover. Also, we are going to adapt the coprocessor to another reconfigurable hardware: The field programmable transistor array (FPTA) developed by our group (see [2]).

We also want to connect several of the proposed systems using the high speed serial links offered by the FPGA. This distributed, autonomous hardware evolution system will give us the resources to tackle large scale optimization problems.

References

1. F. Schürmann, S. Hohmann, J. Schemmel, and K. Meier. Towards an Artificial Neural Network Framework. In Adrian Stoica et.al., editor, *Proceedings of the 2002 NASA/DoD Conference an Evolvable Hardware*, pages 266–273, July 2002.
2. J. Langeheine, K. Meier, and J. Schemmel. Intrinsic Evolution of Quasi DC Solutions for Transistor Level Analog Electronic Circuits Using a CMOS FPTA chip. In *Proceedings of the 2002 NASA/DoD Conference an Evolvable Hardware.*
3. Adrian Stoica et. al. Evolving Circuits in Seconds: Experiments with a Stand-Alone Board-Level Evolvable System. In Adrian Stoica et.al., editor, *Proceedings of the 2002 NASA/DoD Conference an Evolvable Hardware*, pages 67–74, July 2002.
4. Barry Shackleford et. al. A high-performance, pipelined, FPGA-based genetic algorithm machine. *Genetic Programming and Evolvable Machines*, 1(2):33–60, March 2001.
5. J. Schemmel, F. Schürmann, S. Hohmann, and K. Meier. An integrated mixed-mode neural network architecture for megasynapse ANNs. In *Proceedings of the 2002 International Joint Conference on Neural Networks IJCNN'02*, page 2704. IEEE, May 2002.
6. Xilinx, Inc., www.xilinx.com. *Virtex-II Pro Platform FPGA Handbook*, 2002.
7. Micron Technology, Inc., www.micron.com. *Small-Outline DDR SDRAM Module*, Jan 2002.