eDNA: A Bio-Inspired Reconfigurable Hardware Cell Architecture Supporting Self-organisation and Self-healing

Boesen, Michael Reibel; Madsen, Jan

Published in:
Proceedings of the 2009 NASA/ESA Conference on Adaptive Hardware Systems

Link to article, DOI:
10.1109/AHS.2009.22

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

eDNA: A Bio-Inspired Reconfigurable Hardware Cell Architecture Supporting Self-organisation and Self-healing

Michael Reibel Boesen, Jan Madsen
DTU Informatics
Technical University of Denmark
Kgs. Lyngby, Denmark
{*mrb,jan*}*@imm.dtu.dk*

***Abstract*—This paper presents the concept of a biological inspired reconfigurable hardware cell architecture which supports self-organisation and self-healing. Two fundamental processes in biology, namely fertilization-to-birth and cell self-healing have inspired the development of this cell architecture. In biology as well as in our hardware cell architecture it is the DNA which enables these processes. We propose a platform based on the electronic DNA (eDNA) and show through simulation, its capabilities as a new generation of robust reconfigurable hardware platforms. We have created a Java based simulator to simulate our self-organisation and self-healing algorithms and the results obtained from this looks promising.**

***Keywords*-embryonics; self-organisation; self-healing; reconfigurable hardware; biologically inspired;**

I. Introduction

As geometries continues to shrink, the variability increases resulting in an increasing number of both permanent and transient faults, which in turn, increases the demand for robust hardware systems. With current technology, one way to address this problem is to develop hardware that is able to repair itself. The human body and its biological cells is an example of a very robust system. The biological and chemical complexity of the fertilization and later birth is immense - but the basic principles are easily understood. In the fertilization-birth process a new organism is created through cell replication and differentiation. If this self-organising process could be copied to hardware, it would allow individual components of the system to configure and program themselves. A biological cell constitutes the basic programming platform in biology with which a new organism can be ”programmed” and built.

Another biological process which occurs many times every day [1] is the cell self-healing process, in which a dead cell is replaced with a new one of the same kind. For instance the UV radiation from the sun causes some of a persons skincells to die - if they were not replaced the person would at some point become skinless. That is, this process maintains our body, such that we do not die. If this self-healing were implemented in hardware we would have a system that would be able to find and repair faults within itself.

In this paper, we present a DNA structure that can be expressed and interpreted by a, for the purpose built, cell architecture. Previous work assumes that ”hardware DNA” is an FPGA-configuration-bitstring-type of datastructure, where the hardware DNA instructs each particular cell exactly what it will have to do. Our approach views the self-organising and self-healing feature as a part of the same process, thus creating a DNA type which allows the cells (as oppposed to the designer) to autonomously decide where a given functionality expressed by the DNA should be placed. Thus creating a platform which is able to dynamically adapt to any given application and fault-situation and furthermore maintain the system by utilising the descriptive power of our DNA. The proposed cell architecture shows promising results as an underlying hardware platform in resilient systems, where it is critical that the system can recover after failures.

A. Related work

There exists two research branches within biological inspired hardware: Evolvable hardware and embryonics.

The purpose of **evolvable hardware** (EHW) is to use a genetic inspired evolutional model in order to evolve the hardware in question. Researchers working in this area are all working from the Darwinian side of evolution, thinking that the hardware should be evolved through many generations of more or less successful versions of the hardware.

The largest part of the teams working with EHW is concerned with genetic algorithms (for instance [2], [3], [4], [5], [6]). Genetic algorithms can be used to either optimize circuits, to develop them or both. By using a genetic algorithm, research groups have been able to evolve small circuits, such as fx robust multipliers [4]. Others [5] has been able to evolve larger circuits by raising the logical granularity above gate level. Various advantages and drawbacks can be discussed using this technique. Among the advantages is the fact that evolution of such circuits actually is possible using genetic algorithms and the thoroughness with which genetic algorithms explore the search space. In [7] Adrian Thompson succeeded in evolving a tone discrima-

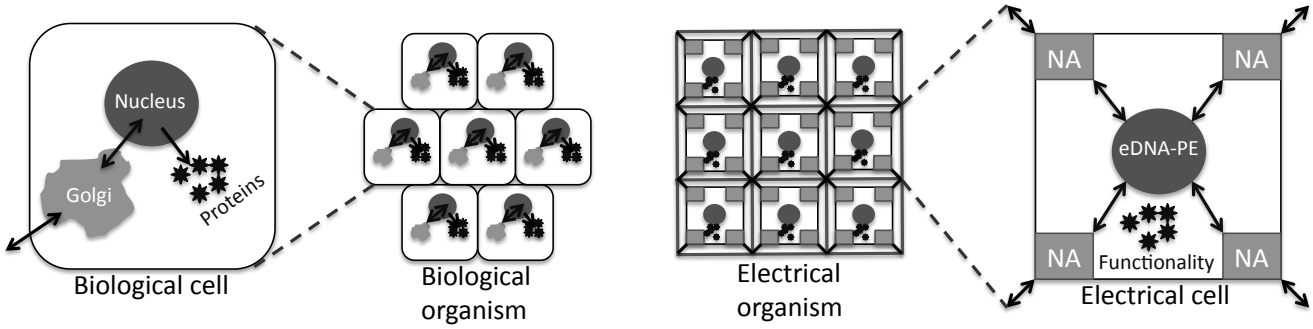

Figure 1. The biological analogy to the system and an overview of the different parts of the system.

tor in a digital (!) FPGA, because the genetic algorithm exploited the physics of the silicon! However, among the drawbacks is the fact that it is primarily an offline approach and it is computational very heavy. Some groups have tried battling this drawback by implementing genetic algorithm operations in hardware (such as [8]). Some have tried to enhance EHW by combining it with simulated annealing [2] and neural networks [6].

Another team have investigated how co-evolution can enhance artificial evolution [9]. Co-evolution is a technique where two or more individuals in a given population battle with each other - a sort of ”arms race”. The ”battling” is implemented by designing a fitness function, which not only considers the fitness of the individual in question but also considers the competing individuals [9]. Another feature of co-evolution is that because the fitness function changes rapidly, the solution which the co-evolutioned systems come up with is not likely to be caught in a local minima. So theoretically, this approach would result in better solutions.

EHW has one major drawback, when the complexity of the target circuit increases, so does the time it take to evolve the circuit, thus EHW cannot (yet) be considered an online approach to adaption.

In **embryonics** researchers are inspired by the biological cell and therefore their work is centered around the creation and configuration of this cell and its components, such as fx the DNA. In [10] the authors have accomplished to make a very robust watch consisting of several cells, which each implement a given function for the watch. If one of the cells dies (from a fault), the watch is able to restore the dead cell and therefore continue operation as normal. When a cell dies the entire column of cells in which the dead cell is located is killed and shifted one position to the right and hereby shifting the part of the organism which is to the right of the dead cell one position to the right.

Yet another group has presented [11] a methodology for how to design a cell-based system which also is highly fault-tolerant. This approach rely on spare cells too, to avoid removing an entire row of cells. But when all spare cells in a row is used and another cell fault occur, the entire row will be eliminated. Both [10], [11] use a DNA which place the functionality for the cells leaving no room for autonomy for the cells and both use a low logical granularity. Others have tried to combine EHW and embryonics [12]. They used the self-reconfigurability of the EHW to repair the fault if possible. If impossible, they program new gates to repair the fault. In 82% of the online test-cases the EHW were able to repair the faults.

Our approach differ from these approaches by using a higher logical granularity and by using a novel DNA type.

B. Introducing the system

Figure 1 shows a sketch of the complete system. It also shows the analogy between a biological cell/organism and our electrical cell (eCell) and electrical organism. The biological organism consists of multiple biological cells as does the electrical organism. The biological cell contains several cell parts (more than is shown in this figure. One which is shown is the golgi apparatus. The golgi is responsible for regulating substances in and out of the cell and forwarding these for processing to the relevant parts of the cell [1]. The eCell contains four such ”golgi” but in hardware these are known as network adapters, which have a similar function. The biological cell nucleus reads and interprets the DNA and from this produces proteins, which defines the function of the cell. Again the eCell have similar parts. The eCell contains an eDNA processing element which reads an interprets the eDNA and from this determines the functionality of the eCell.

Basically, the eCell is the basic programming block in the system. It can be compared to a mix between a CLB of an FPGA and a small CPU. Just as an FPGA has several CLBs, our system contains several eCells. The purpose of each eCell is to read the eDNA and determine what function it is to perform, just like a biological cell do in biology. This will be further described in section II. The eCells can communicate with one another through a communication medium. This medium is a NoC. Currently,

no fixed network topology has been determined. Details concerning the communication medium is purely a NoC-related issue and will not be discussed any further here. Therefore, in this paper we assume that the NoC is a 2D mesh. The eCells contain the eDNA which is our proposed DNA type. It is a programming language which is used to describe the application the user wants to program on the platform. The eCells are capable of interpretting this eDNA and are thus able to program the platform autonomously. This will be further described in section III.

The proposed system has two main features:

1) The **self-organisation** is the process where the eCells read and interpret the eDNA in order to determine what function they have to perform. This means that the eCells know how to partition the eDNA code into several smaller chunks and know how to implement these chunks. This will be further described in section IV.
2) The **self-healing** is the process where if an eCell malfunctions (dies) other eCells detects this and move the functionality which the dead cell had to another functioning cell. This will be further described in section V.

Section VI explains our Java based simulator, which we used to test our system. Finally, our paper will end in describing some results obtained from our Java-based simulator in section VII. In section VIII we will discuss future work with the proposed system and lastly, we will conclude on the work in section IX.

II. eCell: Electronic Cell

The eCell is the fundamental building block of our system. It is the purpose of each eCell to read the eDNA and from this determine what role it is to play in the application. Figure 2 shows a state transition model of the behavior of the eCell and it is described in the following. An eCell is activated once it receives the eDNA. In this case it enters the `Init` state. Here the eDNA is stored in the memory, thus we assume that the local memory of the eCells are large enough. It is not strictly necessary that all eCells contains a copy of the complete eDNA, but they need to access it frequently when doing self-organisation and self-healing, thus it will be beneficial performancewise for all the eCells to a keep copy of the eDNA. Depending on the strategy with which the eDNA is distributed amongst the eCells, the eCell might also forward a copy of the eDNA to another eCell. Several strategies exists; one idea could be to feed the eDNA to the top row of the eCells and then let each eCell forward a copy of the eDNA to the row beneath. Once the initialisation is done the eCell take the transition to the `Differentiated` state. Here the eCell reads and interprets the eDNA in order to determine what function it is to perform - i.e. it performs the self-organisation (see section IV). When it has determined its function it moves on to the `Wait` state. Here the eCell awaits inputs from other eCells or from the environment. Once enough inputs have been received (depending on the function the eCell performs) the eCell moves to the `Executing` state, in which it executes the function it differentiated to in the `Differentiated` state. It also sends the result of this execution to relevant eCells (which also were determined during the `Differentiated` state). Once this result has been sent it returns to the `Wait` state again. Only one more state remains and this is the `Death` state. In this state the eCell is dead and can never recover. The eCell goes to the `Death` state once an error is discovered by a built-in self-test or data integrity test, both of which is based on wellknown techniques and thus not discused any futher in this paper. Furthermore, from all states (except `Death`) it can get to the `Init` state again by resetting itself. This occurs whenever the user of the platform chooses to reprogram the complete platform. Note, that the eCell also can get to the `Init` state whenever an eCell receives information about the death of an eCell. The reason behind this can be found in section V.

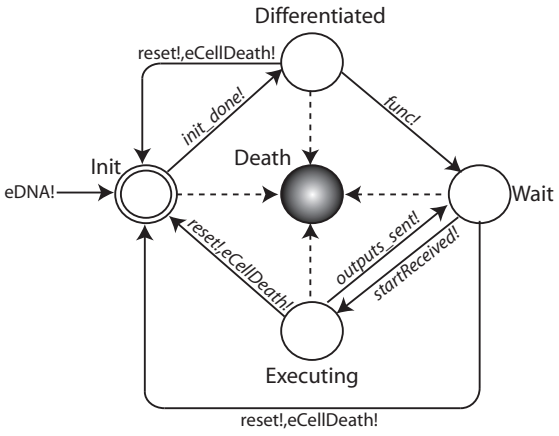

Figure 2. State transistion model of the behavior of the eCell.

III. The eDNA: Electronic DNA

This section will present the DNA of our system - the eDNA. Our eDNA is in fact a programming language and this language is shown in BNF notation below.

```
dna        ::= <statement>* |  <parallel>*
statement  ::= <assignment> | <while> | <if> |
               return <var> | <parallel>
parallel   ::= parallel <statement>* endparallel
assignment ::= <var> = <exp>
while      ::= while <bexp> do
               <statement>* endwhile
if         ::= if <bexp> then
              <statement>* else
              <statement>* endif
exp        ::= <var/c> [<op> <exp>]*
bexp       ::= <var/c> [<bop> <bexp>]*
op         ::= AND | OR  | + | - | ...
bop        ::= AND | OR | < | <= | == | != | ...
```

```
var         ::= Letters{A-Z}* | RAM <var/c>
var/c       ::= Letters{A-Z}* | <const>
const       ::= 0<const>* | 1<const>*
```

The eDNA programming language resembles what can be referred to as ”pseudocode”. That is, it does not allow you to describe your application in great detail, but rather describes the behavior of the application you want to implement. It do allow you to use standard programming features such as variables, array operations, branches and conditional loops. Parallelism in your application will have to be marked manually by using the `parallel` and `end parallel` keywords.

Furthermore, we define a *gene* as a keyword, variable or an operator in this eDNA and these genes are numbered chronologically. This means that if your application described by this eDNA only contains the expression `Z = a` then gene number 1 would be the `Z`, gene number 2 would be the = and gene number 3 would be the `a`. We further elaborate on the gene-concept by defining a gene as ”expressable” if it is

- One of the keywords: `While, if, endif`.
- The first occurence of `RAM`.
- Any operator (except the assignment ”equal sign”).

The eCells are able to interpret this behavioral description and implement the corresponding hardware functionality - this is called the self-organisation (see section IV).

The way the ”behavior-to-hardware” translation is performed is inspired by work done by Ian Page described in [13]. In [13] Ian Page proposed a way to translate software code directly to hardware, by introducing hardware blocks which implements the same functionality as some typical software code constructions. We will use the same basic blocks, but with some minor alterations, which are described in [14]. The resulting hardware blocks, the modifications and what software structures they implement is shown in figure 3(a)-(d). Each figure displays a `start/finish` signal. The `start/finish` signal provides a sequencing mechanism to the system. This means that the order of operation of the software code are maintained by the signal. The signal activates a given block. As seen on figure 3(b) and (c) it is clear that the if and while block respectively is inactive as long as the `start` signal coming from the preceding block is low.

Based on figure 3, figure 4 illustrates the problem the eCells face. They have some building blocks which they, according to the eDNA, have to place on the cell architecture - like a jigsaw puzzle. The next section will explain how the eCells solve this problem.

IV. Self-organisation

The purpose of the self-organisation for each eCell is to interpret the function of the eCell from the eDNA. The self-organisation consists of four steps:

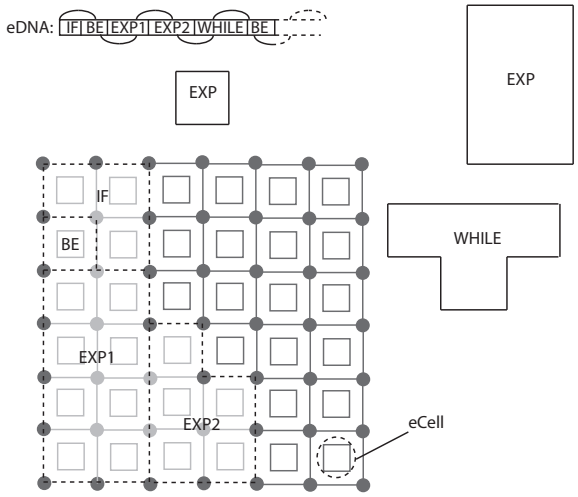

Figure 4. The problem of placing functionality on the cell architecture.

1) `Compute_Cnr`: Determine the eCell number of the eCell.
2) `Find_gene`: From the eDNA, determine the function the eCell has to perform.
3) `Find_outputs`: From the eDNA, determine which eCells the eCell shall send its outputs to.
4) `Find_GS_source`: From the eDNA, determine whether the `start`-signal of the eCell is coming from the environment outside the chip.

The `Compute_Cnr` calculates

$$C_{nr} = ID - DCBC \tag{1}$$

Where the ID is a constant defined either by the user or hardwired in hardware and the $DCBC$ (DeadCellsBeforeCell) is the number of dead eCells which are located ”before” the eCell in question. Now in order to understand what ’before” means we define the spatial relations for eCells.

Definition 1 An eCell X with an eCell number C_X is **before** another eCell Y with eCell number C_Y iff $C_X < C_Y$.

Definition 2 An eCell X with an eCell number C_X is **after** another eCell Y with eCell number C_Y iff $C_X > C_Y$

It is also necessary to realise that

Definition 3 All non-dead eCells have a unique eCell number, which (via an updateable table) referes to their position in the NoC.

Each time an eCell dies the $DCBC$ is being updated thus causes some eCell numbers to change (see section V for more about self-healing).

`Find_gene` makes the eCell search the eDNA for expressable genes. Note from the eDNA syntax in section III, that not all genes are expressable. The algorithm describing how the `Find_gene` work is shown below

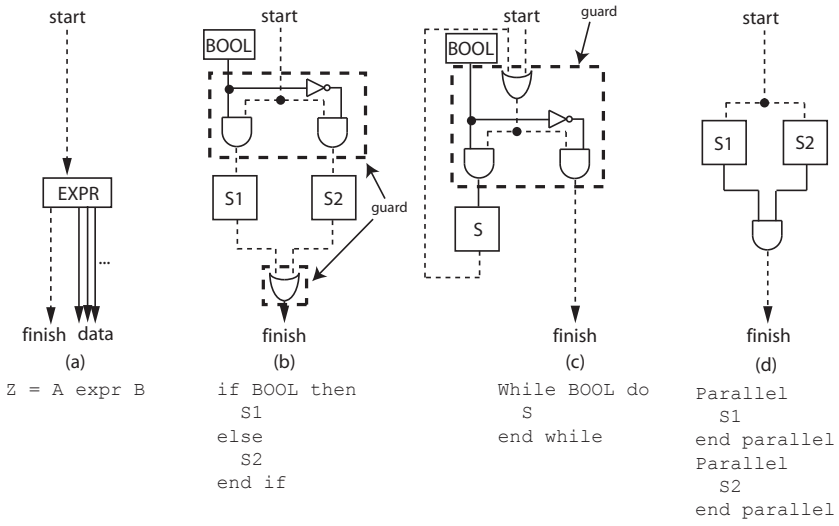

Figure 3. (a)-(c) Modified SW→HW blocks inspired by Ian Page [13]. (d) The parallel block introduced with the eDNA, which is not a part of [13].

```
int Gnr = 0
boolean GeneFound = false;
while (!GeneFound) do
   Gene g = eDNA.getGene(Gnr);
   if (GeneIsActive(g)
       && Gnr == getCellNumber()) then
      GeneFound = true;
      Func = g.getFunc();
   else
      gnr++;
   end if
end while;
```

The `Find_gene` works in a way such that it assigns the functionality of the code to eCells in chronological order. That is the first expressable gene is implemented by the eCell with eCellnumber = 1, the second expressable gene is implemented by the eCell with eCellnumber = 2, and so on. This property makes the job of the `Find_outputs` simple, because the purpose of it is to analyze the algorithm described by the eDNA and deduct the correct eCell numbers. That is, the `Find_outputs` link the different blocks from figure 3 with each other, this is simply done by counting genes in the eDNA. By keeping to this simple chronological order it is easy to link eCells with each other, because the eCells can calculate the position of a given functionality by counting expressable genes in the eDNA. Note, that for eCells responsible for assignments this means that they will need to calculate the position of all eCells which needs the variable they assign. For instance if an eCell has determined through the `Find_gene` that it shall compute the value of `Z = a + b`, then it needs to locate all eCells which uses this `Z`.

Finally, `Find_GS_source` makes the eCell search the genes just before the gene it is supposed to express. If there are no genes before this one in the eDNA, then it requests the input from the environment. This concludes the self-organisation.

V. Self-healing

The self-organisation is designed in such a way, that the self-healing process uses the same algorithms as the self-organisation does. Refer to figure 5 for clarification concerning the following. What happens is, that if a fault is found the routers of the NoC alert the environment to this (through a watch-dog mechanism), which then sends an eCell death-signal to all eCells, notifying that an eCell death has occurred. This causes the eCells to go to their `Init` state (fig. 2) thus restarting the self-organisation. Because the eCells entered the `Init` state by getting a eCell death-signal, they calculate whether they were before or after the dead eCell. If an eCell is located **after** the dead eCell it simply increment the $DCBC$ counter (which is used by the `Find_gene` function in the self-organisation algorithm - see section IV). Updating the $DCBC$ counter has the effect, that eCells which are located **after** the dead eCell automatically gets an eCell number which is one less than the one it had before (see equation 1 and figure 5). This means that it will get the function as its neighbor just one eCell behind got, thus copying itself in a simple, fast and cheap way. However, just before restarting the self-organisation algorithm, it forwards any inputs it has received to the next non-dead eCell located just **after**, because it knows it will move to this position after the self-organisation algorithm has run. If an eCell is located **before** the dead eCell it just checks whether eCells it is sending its outputs to has moved. This is done by compairing these to the dead

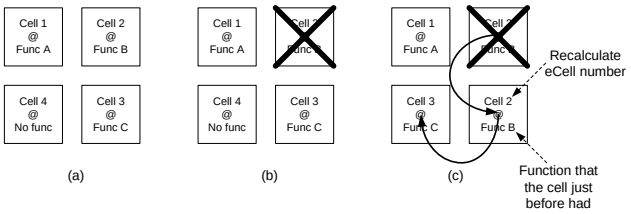

Figure 5. The self-healing process in a small system. (a) The system works as expected, (b) Cell number 2 dies and (c) the self-healing process.

eCell. If the eCells it is targeting is located after the dead eCell then it updates the location of these eCells - and by this the system has healed itself!

VI. The simulator

In order to validate and investigate the algorithms for self-organisation and self-healing we wrote a simulator in Java. The main feature of the simulator were to provide us with a fast method to examine the behavior of the algorithms we invented. All algorithms were based on the idea that they should run on the eCells which used a NoC as communication medium, thus we created the simulator in a way, such that the algorithms could be tested in a ”plug-and-play” fashion. This means that no matter what set of algorithms we tested, the eCells and the NoC stayed the same. Only the behavior of the eCells changed as a result of the algorithms. Therefore in this section we will describe how the underlying architecture of the simulator works. In the following we distinguish between two types of algorithms. The algorithms which the user would like the eCells to implement are called the test-application and the self-organisation and self-healing algorithms are named accordingly.

The simulator has two main features; (1) the user should be able to visually confirm that the self-organisation algorithms and self-healing algorithms are behaving as anticipated and (2) the user should be able to get the results of the test-application such that the user is able to confirm that the system calculates correctly. The first feature requires that the simulator has a GUI and that the rate at which the data in the simulator changes is at a level such that the user has time to confirm at runtime, that the algorithms are behaving correctly. The second feature requires that the system is able to send the results out of the system. This is done by using the `<return>` part of the eDNA language. Whenever the user writes a return statement, the returned variable is ”sent out” of the system to the ”output-environment”. The system has an input-environment and an output-environment. The input-environment is responsible for sending the global start signal to the right cell (which is determined during the `Find_GS_source` part of the self-organisation algorithm) and the output-environment is responsible for receiving the outputs and putting them in a file such the user can access the outputs when the simulation of the test-application is done.

A. Timing & synchronization

The routers being the ones which relays packets between source and destination are therefore also the ones responsible for the timing and the synchronization of the simulator. Each router is instanced as a thread. The behavior of the routers are quite simple. First all routers synchronize around a monitor and then they increment the number of time units passed by one. Then they check all incoming links in the order; north, east, south, west for new packets. If new packets are found they examine the header of each of these and sends the packet in the right direction using dimension order routing. They repeat this forever. But in order for the user to be able to visually confirm that the algorithms work as anticipated the routers sleep for a short while before synchronizing around the monitor. In this way a packet which is being sent between two routers (or a router and an eCell) will pause in a link for a short while thus enabling the user to see the packet being sent between routers and eCells. Furthermore, when a packet flows through a link, the link is colored in a specific color corresponding to the type of packet being sent. In this way the user is able to quickly see exactly what is going on. Once the cells has reached the `Differentiated` state they also visually display which function they perform and by clicking on an eCell the user can access more data about a given eCell such as for instance view the which the eCell is connected to through the NoC.

If a packet has reached the right destination the router sends the packet to the network adapter of the eCell. The network adapter forwards the packet to the eDNA Processing Unit (eDNA-PU) which according to the header activates the algorithm associated with the packet type. Many of the packets an eCell receives requires the eCell to formulate a response packet and since each router synchronize just before repeating their loop again it means that no matter how long the activated algorithm is, the eCell will complete it in between this time. This gives an ”unbalanced” timing as seen in figure 6. This is of course not realistic but was done

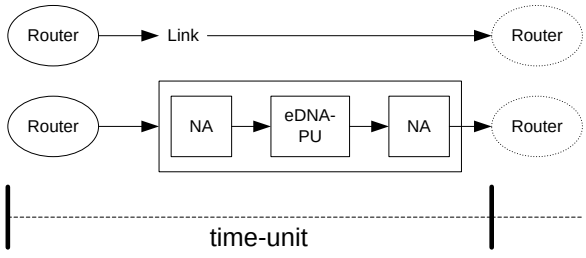

Figure 6. The unbalanced timing of the simulator.

this way in order to make it more simpler to ”plug-and-play”

algorithms. This has no impact on our self-organisation and self-healing algorithms as long as we do not specify any timing constraints concerning how fast the eCell are able to respond to something. And as seen for the self-organisation algorithm (section IV) and self-healing algorithm (section V), no such timing constraints exists. But it is very important to realise this fact since if the system were to be build in hardware the eCells would not be able to respond as quickly as is simulated in the simulator.

VII. Results

Through simulations, we demonstrated that the algorithms described in this paper works as expected. Futhermore our simulator shows that the algorithms described in this paper works for larger examples such as the implementation of the AES, CORDIC and similar algorithms. With the simulator we investigated the three most important statistics of the system; self-organisation speed, self-healing speed and application execution speed. The performance unit for these statistics is called a "time-unit" and is defined as the latency from a router through a network adapter, to the ALU in the eDNA-PU and back to the network adapter (see figure 6). Table I shows all results obtained for our simulator. The applications simulated were the Greatest Common Divisor (GCD) algorithm, Fibonacci number calculator (FIB), CO-ordinate Rotation DIgital Computer (CORDIC) algorithm and AES encryption (only key expansion part). These self-

	GCD	FIB	CORDIC	AES
#cells	9	11	21	88
S-O Speed	0.225	0.250	0.230	0.192
S-H Speed	3	3	4	8
Exe. Speed	97	168	1396	31419
RL	1.916	1.777	3.500	5.034
eDNA size	22 B	29 B	56 B	228 B

Table I
Results from our simulator. The abbreviations translates to: S-O: Self-organisation, S-H: Self-healing, Exe: Execution speed and RL: Routing length.

organisation speed results shows as expected that the NoC is degrading the performance. If the NoC had no influence at all the results would have been 1 new eCell pr. time unit. We can see that the number of eCells needed for each of these algorithms vary a lot and even though the results are pretty much the same. This is also as expected since, because we would expect the NoC to punish the speed pr. eCell equally.

The self-healing speed results were obtained by killing the eCell with eCell number equal to $\lceil \#eCells/2 \rceil$, by being consistent with this we should be able to compare the GCD, FIB, CORDIC and AES. Speed was computed by counting the number of time units until all eCells were finished copying themselves. The results shows that when applications get bigger, the time to repair increases. This is also as expected, because the bigger the application becomes the more eCells have to become aware of the eCell death.

The last statistics measured is the execution speed, that is the time from the algorithm start executing and until it is finished (note, that this of course is dependent on the inputs, so here we just used arbitrary inputs). The execution speed is measured from the time where the eCells are done self-organising. The execution speed results can be used to show what happens when the size and complexity of an algorithm increases. It is definitely anticipated that the execution time will increase the bigger the system becomes and this is also what the results show. This is because the bigger it becomes the more communication time is "wasted" in the NoC. This can also be seen from routing length results. The routing length results explains the higher execution times for the CORDIC and AES algorithms. The results says that each time an eCell in the AES algorithm wants to send its output to another eCell it will on average have to travel approximately 5 hops to reach its destination, thus on average 5 time-units pr. communication is "wasted" in the NoC (recall fig. 6). This gives us two hints; (1) The way the current eCells are numbered initially may need revising (that is the ID in equation 1 section IV may need to be distributed in another way than it currently is) and (2) when the user wants to implement bigger algorithms each eCell may have to implement more than one expressable gene.

Finally, to give an idea of how much local memory the eCells need to store the complete eDNA the final row of table I shows the size in bytes of the eDNA for the applications. Each keyword in the eDNA gets translated to a sequence of bits and since we in the current implementation has 23 keywords each gene in the eDNA occupies 5 bits (because each needs its own encoding). In addition to this come the variables, which might increase the gene size, depending on the number of variables. It is clearly seen that the eDNA is quite compact.

Since the eCell is not implemented as a hardware model, the simulator cannot (yet) provide us with any performance measures, but we created a block diagram of how the eCell could look like in hardware and thus were able to calculate an estimation of how many gates the eCell would consist of. We then divided this number with the number of gates needed to create the functionality which the eCell could interpret from the eDNA in order to get a measure of wasted gates pr. implemented gate (fx if an eCell can be created by 200 gates, and it can implement a function of up to 10 gates, then we have wasted 20 gates pr. implemented gate). We did the same for a Xilinx Virtex 4 CLB and it turns out that this preliminary cost estimation says that our eCell wastes approximately twice as many gates as a Virtex 4 CLB.

VIII. Future work

The most important aspect to acquire some concrete performance measures, i.e. how fast is the system able to

self-organise, self-heal? How much power does it need etc. According to the results in table I it will also be necessary to look into optimizing the NoC in order to save reduce the latency in the NoC. Currently only a simple 2D-mesh is used, other topologies may prove to be better. Thus we need to implement the system as a hardware model and this will be the next step of our work. Another important aspect to illuminate is how the eCell numbers should be distributed initially.

A detailed fault-tolerance analysis will also be interesting to make, once we have a hardware model up and running. It will be valuable to be able to give a clear view of how robust the system is.

IX. Conclusion

This work presents the concept of a model of a new generation of reconfigurable platforms, which also has self-organising and self-healing features. The platform consists of a number of undifferentiated eCells. In order to program the eCells an eDNA is loaded to the eCells. The eDNA proposed gives (unlike other work within this field) the eCells much freedom, because the eDNA describes the behavior of the circuit to be implemented with a small programming language, leaving the interpretation and implementation to the eCells. The eCells interpret and translate the eDNA into functional blocks and then differentiates into these functions - thus they are self-organised. Self-healing has also been implemented. The self-healing was implemented by updating the eCell numbers of the eCells (whenever an eCell dies) and then rerun the self-organisation algorithm. Thus creating a reconfigurable platform which is robust and adaptable.

Acknowledgment

This work was supported by DaNES (Danish National Advanced Technology Foundation) and ArtistDesign (FP7 Network-of-Excellence, 214373).

References

[1] B. Alberts, A. Johnson, J. Lewis, M. Raff, K. Roberts, and P. Walter, *Molecular Biology of the Cell*, 4th ed. Garland Science, 2002.

[2] H. Guo-liang, L. Yuan-xiang, and L. Feng, "Design of evolvable hardware using adaptive simulated annealing," *Proceedings. 2005 International Conference on Wireless Communications, Networking and Mobile Computing, 2005.*, vol. 2, pp. 1390–1392, 2005.

[3] K. Glette, J. Torresen, T. Gruber, B. Sick, P. Kaufmann, and M. Platzner, "Comparing evolvable hardware to conventional classifiers for electromyographic prosthetic hand control," *2008 NASA/ESA Conference on Adaptive Hardware and Systems*, pp. 32–39, 2008.

[4] H. Liu, J. Miller, and A. Tyrrell, "Intrinsic evolvable hardware implementation of a robust biological development model for digital systems," *2005 NASA/DoD Conference on Evolvable Hardware (EH'05)*, pp. 87–92, 2005.

[5] B. Karunya and R. Uma, "Functional level implementation of evolvable hardware using genetic algorithms," *Proceedings of the International Conference Mixed Design of Integrated Circuits and System, 2006. MIXDES 2006.*, pp. 671–674, 2006.

[6] P. Subbiah and B. Ramamurthy, "The study of fault tolerant system design using complete evolution hardware," *2005 IEEE International Conference on Granular Computing*, vol. 2, pp. 642–645 Vol. 2, 2005.

[7] A. Thompson, "An evolved circuit, intrinsic in silicon, entwined with physics," *Evolvable Systems: From Biology to Hardware. First International Conference, ICES96. Proceedings*, pp. 390–405, 1997.

[8] M. Iwata, I. Kajitani, Y. Liu, N. Kajihara, and T. Higuchi, "Implementation of a gate-level evolvable hardware chip," *Evolvable Systems: From Biology to Hardware: 4th International Conference, ICES 2001 Tokyo, Japan, October 3-5, 2001, Proceedings*, p. 38, 2001.

[9] P. Husbands, J.-A. Meyer, and D. Floreano, "How coevolution can enhance the adaptive power of artificial evolution: implications for evolutionary robotics," *Evolutionary Robotics. First European Workshop, EvoRobot98. Proceedings*, pp. 22–38, 1998.

[10] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti, "Toward robust integrated circuits: The embryonics approach," *Proceedings of the IEEE*, vol. 88, no. 4, pp. 516–543, 2000.

[11] T. Plaks, X. Zhang, G. Dragffy, A. Pipe, N. Gunton, and Q. Zhu, "A reconfigurable self-healing embryonic cell architecture," *International Conference on Engineering of Reconfigurable Systems and Algorithms - ERSA'03*, pp. 134–40, 2003.

[12] V. Sahni and V. Pyara, "An embryonic approach to reliable digital instrumentation based on evolvable hardware," *IEEE Transactions on Instrumentation and Measurement*, vol. 52, no. 6, pp. 1696–1702, 2003.

[13] I. Page, "Constructing hardware-software systems from a single description," *Journal of VLSI Signal Processing*, no. 12, pp. 87–107, 1996.

[14] M. R. Boesen, "A model of bio-inspired hardware - the dna approach," *IMM-DTU - Master Thesis*, pp. 1–158, 2008.