“The XILINX Club”

A World Survey of Evolvable Hardware Research Projects Using Xilinx’s Programmable FPGA Chips

Hugo de Garis 1, Andy Tyrrell 2, Pauline Haddow 3, Nicholas Macias 4, Lisa Durbeck 4, John Koza 5, Andrés Upegui 6, Carlos Andrés Peña-Reyes 6, Eduardo Sánchez 6, Lukas Sekanina 7, Esther Hughes 8, Sharon Graves 8, Tim Gordon 9 Srivallaba Mysore 10, Yuya Sasaki 11, Ravichandra Sriram 12

1 Brain Builder Group, Departments of Computer Science & Physics, Utah State University, Logan, UT 84322, USA. degaris@cs.usu.edu, http://www.cs.usu.edu/degaris


3 Department of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway. Pauline@idi.ntnu.no, http://www.idi.ntnu.no/~pauline

4 Cell Matrix Corporation, Blacksburg, VA 24060, USA. nmacias@cellmatrix.com, http://www.cellmatrix.com

5 Departments of Medicine and Electrical Engineering, Stanford University, Stanford, CA, USA.
koza@stanford.edu, http://www.genetic-programming.com/johnkoza.html

6 Swiss Federal Institute of Technology, Logic Systems Laboratory, 1015 Lausanne, Switzerland.
(andres.upegui, carlos.pena, eduardo.sanchez)@epfl.ch,
http://lslwww.epfl.ch

7 Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic.
sekanina@fit.vutbr.cz, http://www.fit.vutbr.cz/~sekanina

8 NASA Langley Research Center, Hampton, VA 23681-2199, USA.
Sharon.S.Graves@nasa.gov

9 Department of Computer Science, University College, London, WC1E 6BT, UK.
t.gordon@cs.ucl.ac.uk,
http://www.cs.ucl.ac.uk/staff/T.Gordon

10 Electronic & Computer Engineering Dept, Utah State University, Logan, UT 84322, USA.
srivallabam@cc.usu.edu

11 Department of Environment and Society, Department of Economics, Utah State University, Logan, UT 84322, USA.
slk1r@cc.usu.edu

12 Brain Builder Group, Department of Computer Science, Utah State University, Logan, UT 84322, USA.
ravichandras@cc.usu.edu

=================================

Abstract

This collection of 9 short papers (typically of 4-6 pages each) aims to provide an overview of state-of-the-art research in the new field of Evolvable Hardware (EH). These papers have at least one aspect in common, namely that they all report on use of Xilinx’s FPGA (field programmable gate array) chips as their experimental platform to a greater or lesser extent. This collection includes recent work from some of the best researchers in the field, and thus hopefully will give readers a comprehensive feel for where EH is situated currently, and where it is headed.

=================================

Introduction

The research field of evolvable hardware began (arguably) in the early 1990s when the first author had the idea that the configuring bitstring of a programmable chip might be conceived as a chromosome of a Genetic Algorithm (GA), and began to propagate the idea. It followed that if a randomly generated configuring bit string was receivable by the chip without blowing it up (as is the case with chips
based on RAM technology) then a random configuring bitstring (chromosome) would generate a randomly configured circuit, whose quality of performance at a given task could be measured and given a number (or a “fitness” value, to use the GA terminology). If one had another circuit, conventionally programmed to measure the fitness of the evolving circuit, then the genetic algorithmic evolution of electronic hardware, all happening at hardware speeds (i.e. typically 100s to 1000s of times faster than conventional PC based software speeds) would become possible. There are two broad categories of evolvable hardware -

a) where the evolution occurs in software simulation, at software speeds, labeled by the first author in 1992 as “extrinsic” evolvable hardware, i.e. the evolution is occurring outside the chip, in the simulating computer.

b) where the evolution occurs directly in the programmable chip, at hardware speeds, labeled by the first author, “intrinsic” evolvable hardware, i.e. the evolution is occurring inside the chip.

Over the course of the past decade, a shift in emphasis has been occurring in which a growing proportion of EH research work has become intrinsic. Some of the main themes in the EH field have become –

a) the attempt to evolve robust circuits, which are more immune to fabrication faults. Some of the papers in
this collection deal with this topic. As molecular scale circuitry becomes increasingly common in the next decade, it will become statistically virtually impossible to fabricate circuits with trillions of components that are perfect. Hence an EH based approach to circuit generation, that is also robust is very attractive.

b) the attempt to evolve digital and analog electronic circuits that perform useful functions, especially if they perform at or beyond human designer level of competence.

c) the attempt to wed ideas from biology and electronics, for example to have electronic components grow, differentiate, and self repair, in a form of “electronic embryology” or as it is known increasingly “embryonics”.

The above themes, and others, are evident in the papers that follow, whose titles, authors, and affiliations, are now listed. As one can see below, 5 of these 9 papers come from Europe and 4 from the USA, showing that the EH research field is fairly evenly developed on both sides of the Atlantic. Hopefully, other regions of the world will play a greater role in the development of EH in future years.

Paper 1, entitled “Intrinsic Hardware Evolution and Fault Tolerance”, is by Andy Tyrrell, of the University of York, UK, Europe.
Paper 2, entitled “Evolution and Development Friendly Technology and Techniques for Achieving Complex and Adaptive Circuits”, is by Pauline Haddow, of the Norwegian University of Science and Technology, Trondheim, Norway, Europe.

Paper 3, entitled “Cell Matrix™ Demo Boards for Evolvable Hardware Research”, is by Nicholas Macias and Lisa Durbeck, of the Cell Matrix Corporation, Blacksburg, VA, USA.

Paper 4, entitled “Evolvable Hardware and Rapidly Reconfigurable Field Programmable Gate Arrays”, is by John Koza of Stanford University, Stanford, CA, USA.


Paper 6, entitled “From Evolutionary Design of Image Filters to Virtual Reconfigurable Circuits and Evolvable IP Cores”, is by Lukas Sekanja, of Brno University of Technology, Brno, Czech Republic, Europe.

Paper 7, entitled “Evolvable Hardware for Aerospace Signal Processing Applications”, is by Esther Hughes and
Sharon Graves, of the NASA Langley Research Center, Hampton, VA, USA.

Paper 8, entitled “Evolvable Hardware and Biological Development”, is by Tim Gordon, of University College, London, UK, Europe.

Paper 9, entitled “The e-LEM Project”, is by Hugo de Garis et al, of Utah State University, Logan, Utah, USA.

---

Paper 1

“Intrinsic Hardware Evolution and Fault Tolerance”

Andy Tyrrell

Bio-Inspired & Bio-Medical Engineering, Department of Electronics, University of York, UK.

Andy.Tyrrell@bioinspired.com,

http://www.bioinspired.com

---

Introduction

Fault tolerance is a technique used in the design and implementation of dependable computing systems. With the increase in complexity of systems, complete fault coverage at the testing phase of the design cycle is very difficult to achieve, if not impossible. In addition, environmental effects
such as Radio Frequency and Electromagnetic interference and misuse by users can mean that faults will occur in systems. These faults will cause errors and if left untreated, could cause system failure. The role of fault tolerance is to deal with errors, caused by faults, before they lead to failure.

The majority of hardware redundancy reconfiguration techniques rely on complex algorithms to re-assign physical resources to the elements of the logical array. In most cases these algorithms are executed by a central controller that also performs diagnosis functions and accomplishes the reconfiguration of the physical system [1,2]. This approach has been demonstrated to be effective, but its centralised nature makes it prone to collapse if the processor in control fails. These mechanisms also rely on the designer making a priori decisions on reconfiguration strategies and data/code movement, which are prone to error and may in practice be less than ideal. Furthermore, the timing of signals involved in the global control are often prohibitively long and therefore, unsuitable for real-time fault tolerance.

Traditionally, fault tolerance has been added explicitly to system designs by including redundant hardware and/or software which will "take over" when an error has been detected. A novel alternative approach would be to design the system in such a way that the redundancy was incorporated implicitly into the hardware and/or software during the design phase [3,4]. This should provide a more holistic approach to the design process. We already know that Genetic Algorithms and Programming can adapt and optimise their behaviour and structure to perform a specific
task, but the aim is that they should “learn” to deal with faults within their operation space. This implicit redundancy would make the system response invariant to the occurrence of faults.

Xilinx’s Virtex devices now allow us to perform real-time partial reconfiguration on hardware [5]. This section explores the possibilities of using evolutionary techniques to first produce a processing system that will perform a required function, and then consider its applicability for producing useful redundancy that can be made use of in the presence of faults, i.e. is it fault tolerant? The results shown are based around a simple example, an evolved oscillator [6]. All evolution was performed on a Virtex 1000, using JBits.

**Evolved Fault Tolerance**

Faults can be viewed in a system similar to changes in the environment for evolving biological entities. The individuals compete in the new environment to produce a species which is capable of living and reproducing with the new changes. This method of adapting to faults requires a change in the way most evolutionists view evolutionary algorithms.

In general, evolutionary algorithms are viewed as an optimising process, where the algorithm is halted when some maximal/average fitness is reached. For evolutionary strategies to be used to provide fault tolerance, the algorithm should be viewed as a continuous real-time adaptation
system, constantly adapting to changes in the environment. These can come from a number of sources, for example:

- hardware failures leading to faults;
- requirements/specification changes;
- environmental changes (e.g. EM interference).

When such a change occurs, the fitness of the individuals will change correspondingly. As the evolution progresses, the individuals will adapt to the new changes until they regain their fitness.

Critical to any fault tolerant system is useful redundancy, whether in the form of virtual hardware/software or real hardware/software and possibly design. One of the features of an evolutionary system is its reliance on having a diverse population from which to evolve from. The hypothesis tested in this paper concerns the applicability of these diverse members to provide useful redundancy in the presence of faults, i.e. provide inherent fault tolerance [7,8]. In [7], the authors considered Populational Fault Tolerance, where they affected faults into a number of transistor circuits. In this work high degrees of genetic convergence were achieved and the authors speculated on what might be providing redundancy within a given population, and hence allowing fault tolerant properties to exist. The work reported here encourages genetic convergence not to occur and hence considers a related but different set of evolved circuits.

JBits
The work reported involves the intrinsic evolution on Xilinx’s Virtex devices. This made considerable use of JBits and a brief overview of JBits is now given. JBits [9] is a set of Java classes which provide an Application Program Interface (API) into the Xilinx Virtex FPGA family bitstream. This interface operates on either bitstreams generated by Xilinx design tools, or on bitstreams read back from actual hardware. It provides the capability of designing and dynamically modifying circuits in Xilinx Virtex series FPGA devices. The programming model used by JBits is a two dimensional array of Configurable Logic Blocks (CLBs). Each CLB is referenced by a row and column index, and all configurable resources in the selected CLB may be set or probed.

[6] showed an example application programmed using JBits, the requirement was to change the LookUp Table (LUT) entry so that the function of the gate is changed from an OR gate to an AND gate (a simple example). To perform this application a baseline bitstream is created which contains the required input and outputs to the LUT. The LUT is forced to be in a specific CLB on the FPGA so that it can later be located. Once this has been done the Java

<table>
<thead>
<tr>
<th>Input Vector</th>
<th>OR</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>G3 G2 G1 G0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 0 1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0 0 1 0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 1: Truth Table for AND/OR gates

program as shown in [6] can be written. The program is written to first load in the baseline bitstream, this bitstream is then modified to change the function of the cell (in this case change the OR gate into an AND gate), the LUT entries for the AND and OR gates are calculated by taking the truth table and writing the outputs into a single 16bit (4 nibble) value:

  OR  1111 1111 1111 1110  0xffffe
  AND 1000 0000 0000 0000  0x8000

Once the LUT has been changed (within JBits), the bitstream is then retrieved and sent to the Virtex Chip.
Since the JBits interface can modify the bitstream in this way it is still possible to create a 'bad' configuration bitstream; for this reason it is important to make sure that the circuit which is downloaded to the chip is always a valid design. This is done by only ever modifying a valid design to implement the evolvable hardware. This is a method similar to that demonstrated in Delon Levi’s work on GeneticFPGA in [10] where the XC4000 series devices were programmed to implement evolvable hardware. The major obstacle in that implementation, the speed with which the devices could be programmed, is no longer a problem here.

The hardware interface is the XHWIF (Xilinx Hardware InterFace). This is a Java interface which uses native methods to implement the platform dependent parts of the interface. It is also possible to run an XHWIF server, which receives configuration instructions from a remote computer system and configures the local hardware system. This enables the evolvable hardware to be evaluated in parallel on multiple boards. This interface is also part of Xilinx’s new Internet Reconfigurable Logic (IRL) methodology, where the computer houses a number of Xilinx parts which can easily be reconfigured through the Java interface.

**Evolutionary Strategies and Evolvable Hardware (EHW)**

For simplicity a 16*(1+1) Evolutionary Strategy (ES) was used for the majority of experiments reported. The (1+1)ES simply works by creating a parent, mutating this parent to
create a child and comparing the fitness of the parent and the child. The best individual becomes the parent for the next generation. In the $16^*(1+1)ES$ this process was replicated 16 times, with little interaction between the $16^*(1+1)ES$, allowing increased diversity.

Further selection pressure is created through the overwriting of consistently bad $(1+1)ES$’s by another random $(1+1)ES$. The environment is similar to a number of species evolving, where each $(1+1)ES$ is a species (even though there is only ever one individual!). Each species competes for its fitness with fifteen other species; if a species’ fitness drops due to a change in the environment (which could be a literal change in the environment or a change in the genotype due to a mutation for example), the species will be deleted (xenocide) and replaced with one of the other species, the choice as to which one is used for replacement is done randomly. Although the number of species would seem to decrease with this method, it in fact does not appear to be the case, since once the new species has been in existence for a number of generations it will be significantly different from the other species due to mutations.

All experiments were performed on a Xilinx Virtex FPGA system, which allowed partial reconfiguration to be performed using JBits [9]. The application was to evolve a simple 3KHz oscillator on this device [10]. A fitness level of 200 was set to be an acceptable answer. The hardware configuration is shown in figure 1. Rather than checking the actual frequency of the signal by polling the output value and checking to see when it is high and when it is low, the
fitness is evaluated by including a counter that counts the number of positive edges within a certain time period. This proved to be a more accurate method of frequency measurement.

The results for three different mutation rates are shown in figures 2-4. The fault model used to test the fault tolerance of the system was one that considered only stuck-at-faults (stuck-at-one - SA1 and stuck-at-zero - SA0) and short circuits. These faults were emulated and inserted into the bit-stream of the device. The position and fault type were specified by the setting of particular LUTs within a device. Further faults were introduced into the circuit until, in most cases, the system failed.

**Results**

An important measure when implementing the fault tolerant hardware is the rate of mutation implemented in the 16*(1+1)ES. The measure by which the rate of mutation is decided is the rate of change of the genotype, and in a related way the disruption created by the mutation operator. These two measures are orthogonal to each other; the higher the mutation the faster the genotypes can respond to change (faults) but the higher the mutation the more disruption caused to the genotype. With low mutation rates the system becomes less responsive but is more stable once a good fitness has been reached. Three mutation rates are shown in figures 2-4, high mutation level (5 mutations/individual/generation), medium level (2
mutation/individual/generation) and low level (1 mutation/individual/generation).

The most striking feature observed from figures 2-4 is that when faults are injected into the best individual there is another individual in the population that can be identified as the new best with a fitness sufficiently high to be acceptable for use, until a sufficient number of faults are injected and the whole population fails. There does appear to be sufficient redundancy produced by the 16*(1+1)ES to provide fault tolerance for these type of faults. A more detailed consideration of the results highlights the different characteristics of the three mutation rates. With low mutation rates the “jitter” in the fitness is low and when faults occur, which require the circuit to evolve around the fault, the regaining of fitness levels is relatively slow. When mutation rates are high, the “jitter” in the best individual is much more pronounced, but with this mutation rate the fitness of the best individual climbs much more rapidly than with a low mutation rate. The medium mutation rate appears to give the best overall response in terms of both these measures. The “jitter” in the fitness is less pronounced than with a high mutation rate, but when a fault occurs the individuals regain fitness more quickly than for low mutation rates.

Another possible measure of diversity would be to actually look at the “circuit” layout of individuals in a population. The problem is how to identify which matrix elements in the FPGA are providing useful functionality and which are not. An experiment was devised to help provide this information.
The genotype is tested by first inserting a Stuck At 0 fault (SA0) and retesting the fitness, inserting a Stuck At 1 fault (SA1) and retesting. Finally, the faults are removed and testing takes place again, just to be sure that the genotype still works even without faults. If the results for the best individual have a fitness that was reduced by at least half, the individual is considered dead and that particular element in the overall matrix is considered critical to the functionality. There is an extra bit (which is not used in the genotype/phenotype map) that describes whether that individual matrix element is used by the circuit. This bit and the corresponding matrices can then be used to build a map of the useful/useless elements within the matrix in the current best solution.

Our results show just how varied the different runs of the 16*(1+1)ES are. Now, as indicated earlier, these results cannot accurately tell us how much of the circuit is being used because there may be redundant parts to the circuit that mean that although they are being used, the fitness does not decrease to half the maximum value when a fault is inserted. Instead, what is noticeable is that the parts of the circuit that are definitely required, are more likely to be damaged through faults than in other parts.

**Comments**

The results presented indicate that when individuals within a population are prevented from having genetic convergence, the population as a whole does present characteristics that might be very useful in safety critical environments.
The work presented here assumes that an initial evolutionary stage has taken place before the system goes “on-line”. This alleviates to some extent the problems of on-line evolution and the fact that many of the mutants are less than fit. This allows for the on-line system to consist on mostly “useful” individuals. In addition, the evolutionary strategy used means that these are relatively unrelated individuals enhancing the probability of useful diversity. Evolution, once on-line, takes place on all individuals apart from the current “best”. This reduces the effect of low fitness outputs caused by mutation.

It seems clear from the experiments that evolutionary strategies can be used to produce redundant versions of systems that increase the fault tolerance of an application. Further work is required to investigate other evolutionary strategies such as Genetic Algorithms, but the results presented here contribute to this exciting and rapidly expanding subject.

References


Figure 1: Virtex hardware set up
Figure 2: Fitness of the best individual when subjected to faults, with a low mutation rate

Figure 3: Fitness of the best individual when subjected to faults, with a medium mutation rate
Paper 2

“Evolution and Development Friendly Technology and Techniques for Achieving Complex and Adaptive Circuits”

Pauline Haddow

Dept. of Computer and Information Science, Norwegian University of Science and Technology, Trondheim, Norway.

Pauline@idi.ntnu.no, http://www.idi.ntnu.no/~pauline

Introduction

The field of evolvable hardware (EHW) is relatively young but already there have been many changes in the technology platforms used. The appearance of Xilinx's 6200 series enabled Thomson [THOM96] to introduce the electronic design community to the idea of evolvable hardware. The 6200 series offered many evolution friendly features such as partial reconfiguration; access to a single CLB; configuration unit of a single bit; open architecture and protection against illegal configurations. This opened for a new and exciting design possibility - evolvable
hardware. Many other researchers have worked on the 6200. However, with the demise of the 6200 series many researchers worked on the 4000 series. The 4000 series is not partially reconfigurable but, at that time, available unlike the promised Virtex series which finally replaced the 6200 series. When Virtex arrived, although offering some of the features of the 6200, not all of the evolution friendly features were retained. As such, not only have researchers had to move through different technology platforms but also features have disappeared.

Some researchers have attacked this problem by building their own platforms [LAY98, MAC99, MUR98, STO02, LMS02, SAN01]. Although these platforms enable progress in evolutionary design techniques, we need a broadly available technology platform for research in this area. A mass produced evolution friendly reconfigurable platform is not likely to be ahead of us and a newer technology more evolution friendly than traditional reconfigurable platforms is not around the corner. To be able to reuse results and lessons learned from today's technology on tomorrow's and exploit the power of evolution, one solution is to provide a virtual evolution friendly reconfigurable platform that may be mapped onto a given technology.

**The Virtual Sblock FPGA**

Reconfigurable technology may be said to be advancing in a more exponential rather than linear rate, with respect to the logic and routing resources offered by successive
generations. As such, today’s FPGAs provide significant resources for the design of more complex electronic circuits – the challenge being how to design and test them. EHW has been proposed as a way to achieve complex electronic design. However, this growth in resources is in fact a limitation to evolving complex designs. In Evolvable Hardware, a one-to-one mapping for the genotype-phenotype transition is often assumed. Use of a one-to-one mapping assumes a genotype consisting of a complete blueprint of the phenotype. For an electronic circuit, the genotype thus completely describes the circuit topology i.e. the components and their interconnections. As such, this dramatic increase in resources also demands a dramatic increase in the complete description of the circuit i.e. large genotypes. A fuller description of the requirements that EHW places on the technology and an introduction to the Virtual Sblock FPGA may be found in [HaTu00].

The virtual Sblock FPGA is designed to provide a virtual platform for EHW experiments. The key goals are to provide a much simpler FPGA - thus reducing the length of the genotype description, and providing a virtual platform that may be mapped onto today’s and tomorrow’s technologies.

The Virtual EHW FPGA contains blocks --- \{Sblocks\}, laid out as a symmetric grid where neighbouring blocks touch one another. There is no external routing between these blocks except for the global clock lines. Input and output ports enable communication between touching neighbours. Each Sblock neighbours onto Sblocks on its 4 sides. Each
Sblock consists of both a simple logic/memory component and routing resources. The Sblock may either be configured as a logic or memory element with direct connections to its 4 neighbours or it may be configured as a routing element to connect one or more neighbours to non-local nodes. By connecting together several nodes as routing elements, longer connections may be realised. As such, one of the most important features of the virtual Sblock FPGA is removal of the vast routing resources available on today’s FPGAs. In addition, the logic blocks themselves - Sblocks, are relatively simple. A fuller description of the virtual Sblock FPGA and its mapping onto Xilinx’s VirtexE may be found in [HaTu01].

**Extrinsic, Intrinsic and Complete Hardware Evolution**

The virtual Sblock FPGA provides a virtual platform for EHW experiments. This platform may either be implemented in software or hardware. When the virtual Sblock FPGA is implemented in software, extrinsic evolution experiments may be conducted. That is, all individuals of the evolutionary process are evaluated in software. For intrinsic EHW experiments, a given individual and the virtual Sblock FPGA are mapped onto the physical FPGA so that the individual may be evaluated using traditional test vectors and the results fed back to the evolution process (implemented in software). For Complete Hardware Evolution, both the evolutionary algorithm and the virtual Sblock FPGA are implemented on the physical FPGA together with the evolving circuit.
In today’s FPGAs, considering the vast amount of logic, memory and routing resources, it may be said that reconfiguration of these devices is relatively fast. However, when you introduce a technique such as evolution, where there may be a need for 1000s or even 100,000s of reconfigurations to evolve a single circuit, suddenly the configuration speed of today’s devices becomes a problem. This problem is especially noticeable for intrinsic evolution.

To achieve intrinsic evolution in a reasonable timeframe, partial configuration is an important feature of today’s FPGAs – although it may be said that little support or documentation is available. Configuring only the elements that have changed since the last configuration can drastically reduce the amount of configuration data and, as such, the time used for reconfiguration.

Interesting work in this area is being conducted by Sekanini [SEK03, SERU00]. Although the current solution suffers from scalability problems, the goal is what might be termed an application specific intrinsic architecture (virtual circuit) which may be downloaded onto today’s FPGAs and support configuration of the evolving design.

**Development Friendly Features of the Virtual Sblock FPGA.**

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. Combining artificial development with evolution thus aims to improve the complexity of electronic circuits that may be evolved. A developmental genotype may be used to describe a circuit for today’s FPGAs and, as such, provide a drastically reduced genotype in relation to the phenotypical description needed. However, today’s FPGAs were not designed with a developmental technique in mind and, as such, are not particularly suitable for a developmental approach.

When we consider development, we may say that the Virtual EHW FPGA is a small step in the right direction for providing a development friendly platform. Artificial development could, as in nature, use: pattern formation, morphogenesis, cell differentiation and growth as interrelated processes. Biologically we are looking at a cell-based organism where the maximum size of our organism is the maximum grid size of the technology we are using. Each Sblock is a cell in the organism and can communicate with other cells through local communication - although limited to 4 neighbours. Cell differentiation may be achieved by reprogramming a given cell to connect or disconnect to given neighbours and programming the cell's internal functionality. It should be noted that programming the cell's connectivity and functionality are intertwined through the look up table. The development algorithm should control pattern formation within the cells based on the cell types i.e. Sblock function described by its Look-up Table (LUT). To achieve morphogenesis, it is possible to
abstract slightly from the technology and not only view the
grid as an organism and view the cell itself as an expanding
body, consisting of one or more cells.

**Current Evaluation Platform for Intrinsic and Complete Hardware Evolution**

The platform consists of a cPCI computer system including
a BenERA FPGA board. The BenERA board includes a
Xilinx VirtexE FPGA that can be configured from within
the cPCI computer system. The FPGA is configured
through the JTAG port (serial) due to little support for the
SelectMap port (parallel). The JTAG port has a maximum
clock speed of 20 MHz. However, the number of
configurations per second that may be achieved is much
less than that which might be expected from this maximum
clock speed. For intrinsic evolution, each individual must
be downloaded onto the physical FPGA and its fitness
assessed. That is, for a typical intrinsic implementation, at
each iteration the virtual Sblock FPGA architecture and the
individual circuit are downloaded. Intuitively, since the
virtual Sblock FPGA is fixed for the complete evolution
process it would be better to download it first and then just
download the individuals on this architecture i.e. a two-
stage process.

Achieving this two-stage process in practice was not so
easy, due to the fact that Xilinx’s software is geared
towards traditional design and the fact that the bit stream
format is not available to the general public. For this
process we needed to be able to embed the virtual Sblock
FPGA on the physical FPGA thus fixing the configuration of a number of resources (both routing and logic), on the physical FPGA such that these would not be changed by the second stage. This functionality is not available through Xilinx’s software tools. In addition, after achieving the embedded architecture, we needed to know where and how to partially reconfigure the physical FPGA. The first solution involved the generation of a Xilinx bit stream for the virtual Sblock FPGA architecture using Xilinx ISE software where CLB slices were instantiated in the VHDL code through the use of hard macros. Once the bit-stream was downloaded, it was read back and then parts of the bit stream were edited to reflect a given individual. The position of the edited bits was found through both available formatting information and reengineering missing information. This process was not without its troubles and although an almost correct solution was found, some errors appeared. The final solution involved the generation of an on-chip bit stream (non Xilinx) formatted for this purpose as well as configuration logic to drive configuration (and partial configuration) of the physical FPGA. More information on this approach may be found in [TuHa04].

**Ongoing Work**

The cPCI platform is being used, both for work related to the virtual Sblock FPGA and developmental techniques and also as a platform for running other EHW experiments on Xilinx Virtex chip(s). To date, work within what has been termed “Messy Gates” [MIHA01] – the use of evolution to achieve fault tolerant architectures, has been conducted
using extrinsic evolution. Ongoing work is using an intrinsic approach on the cPCI system so that features of the messy gates approach may be tested out on a physical platform. In addition, work into the use of evolution towards adaptive solutions is continuing based on both intrinsic and Complete Hardware Evolution approaches on the cPCI system.

References


[MAC99] N. Macias, “The Pig Paradigm: The Design and use of a massively Parallel Fine Grained Self-


---

**Paper 3**

“Cell Matrix™ Demo Boards for Evolvable Hardware Research”

Nicholas J. Macias and Lisa J.K. Durbeck

Cell Matrix Corporation
Blacksburg, VA 24060, USA.

nmacias@cellmatrix.com
www.cellmatrix.com

We are currently developing Cell Matrix™ demo boards, using a Xilinx Spartan-series FPGA. Each board contains an 8x8 Cell Matrix, and can be used to conduct evolvable hardware research. They will be available for purchase by early 2004.

The Xilinx® Spartan™ line of FPGAs is an excellent choice for implementing custom circuitry, because it provides high performance at an exceptionally low price.
Within the Spartan line, we have worked with two devices: the XC2S200, which is a 200,000 gate device, and the XC2S300E, which is a low-power 300,000 gate device. Notable features of the XC2S300E device include:

- 300,000 gates;
- low operating voltage (1.8V);
- available today;
- low cost: around $35 per device;
- easily configured with a single PROM;
- can be made to work with TTL voltage levels;
- convenient packaging (PQFP-208);
- plenty of user I/O (at least 140 lines); and, not the least of all,
- free software (WebPack™)

For our application of this device, there is an additional feature, which is the ability to use the Spartan's internal 16-bit lookup tables (LUTs) as distributed RAM. Combining eight LUTs thus gives us 128 bits of RAM, which is the size necessary for each Cell Matrix cell's internal lookup table. This affords us higher density than, say, creating RAM out of individual flip-flops. Even so, the large amount of distributed memory in a Cell Matrix quickly consumes much of the device resources. Taken together, these features make the Spartan XC2S300E an excellent choice for implementing a Cell Matrix via programmable logic.
The Cell Matrix architecture is independent of any particular implementation. It may be implemented in software as a simulation, in hardware on top of an FPGA, directly in silicon as a custom ASIC chip, or by some method as yet undiscovered, such as novel mechanical, chemical, or biological techniques. We have already created several software versions of the Cell Matrix and have been working with them for a number of years. These are extremely useful for developing and debugging very large circuits. Simulators do not, however, take the place of hardware implementations. Even for small Matrices, a physical system has a number of advantages over a simulated system: it can be incorporated into other physical systems more directly, and it is also more straightforward and direct to interface the high number of I/O lines available on the edges of Cell Matrix hardware to other hardware. Working with real hardware also exposes the physical variations of the implementation to the researcher, sometimes leading to completely unexpected results, as illustrated by the work of Adrian Thompson [REF A].

In some ways, we would have preferred to implement the Cell Matrix directly in silicon, using full custom ASIC. Going with a custom ASIC would have given us a much more compact implementation of our design, and thus would have afforded much higher density -- given our current VLSI layout we estimate that we could fit over 10,000 cells per chip using a 0.18μ process, whereas with the XC2S300E, we can fit slightly more than 100 cells on the device. However, an 8x8 array of cells is enough to do some initial, quite interesting evolvable hardware research,
and we are designing the boards so that a larger Cell Matrix can be set up by simply connecting multiple boards together.

A custom ASIC Cell Matrix is also a more desirable platform for studying fault tolerance and fault handling strategies. This is because the Cell Matrix is architecturally defect tolerant, in the sense that a defect in one cell will usually not impair the entire system. Defects are limited to the damaged cell itself, and perhaps its immediate neighbors. Since we are instead implementing the Cell Matrix on top of another device, fault tolerance and fault detection cannot exceed the degree of fault tolerance of the underlying device. Since the Spartan XC2S300E contains numerous global signals, it is possible that a single defect may impair large parts of the system, regardless of how fault tolerant the configuration above it is.

The economic bottom line for our decision to use an FPGA instead of going ASIC is the reality faced by many hardware companies. Initial costs for a fully custom, 0.18μ, six-layer ASIC process can run into the hundreds of thousands of dollars. With a Spartan FPGA, we get many of the benefits of that technology (high speed, low power), as well as extremely fast design turnaround (minutes instead of months), all for a fraction of the cost.

**Board Details**
The boards are still under development, but the preliminary design specifications for these boards are as follows:

- contains an 8x8 matrix of four-sided cells;
- single power supply required, 6-9V DC;
- single system clock required, with minimal waveform requirements;
- all inputs default to logic 0 if left unconnected;
- boards hook together to seamlessly form larger matrices;
- cost: under $300;
- can operate in a standalone mode;
- built-in support for interfacing from a PC

All edge cells along the perimeter of the 8x8 matrix will have their edge inputs and outputs accessible via edge connectors around the perimeter of the board. I/O will be TTL compatible. Additionally, it will be easy to connect two or more boards so that the edge cells' D and C mode inputs connect, thereby forming a larger matrix. This tiling may be done to arbitrary dimensions (e.g. 16x8, 8x16, 8x32, 16x16, and so on). It should also be possible to create fully functional tilings that contain holes.

These boards will be fully functional in stand-alone mode, i.e., they do not need to be hooked up to a PC, neither initially nor while running. However, an interface card will be available to allow the board to be controlled and monitored by a PC. This will allow one to load configurations into the matrix from the connected side. The interface will allow one to read all edge cells' outputs (C
and D lines). The interface will also allow one to write to the C and D inputs of any edge cell that is not connected to another board. The system clock and reset lines can also be controlled via this interface. We are currently working on a parallel port interface, though we are looking at high-speed USB and Ethernet interfaces as well. The interface can be used to configure a larger, multi-board matrix as well as a single board matrix.

We are aware that research funding is always tight, and we are working to keep the cost of these boards low. Our current estimate for the purchase price is under $300. We also intend to make some boards accessible through a web interface for undergraduate projects, unfunded research, and other projects that cannot afford to purchase their own.

**Evolvable Hardware Support**

We encourage the EHW community to use these boards for EHW research, and also to use the freely available Cell Matrix simulators [REF D]. There are a number of ways in which the Cell Matrix is highly suitable for evolvable hardware. The architectural details are openly and readily available. This means no proprietary knowledge is necessary in order to work with the board at a low level. Groups can develop their own software for configuring the board, or their own hardware for interfacing to it. This not only suits the do-it-yourself attitude of many in the EHW community, but is also highly beneficial in learning situations, such as university lab classes.
Cell Matrix hardware is not susceptible to damage due to improper configuration. Completely random configuration information can be safely loaded into the hardware, thus allowing one to directly evolve and mutate configuration strings. This is an extremely desirable feature for EHW work, and is one of the reasons for the former popularity of the now-extinct 6200-series Xilinx devices [REF B]. The Cell Matrix cannot be damaged by "incorrect" configuration mainly because it is impossible to connect outputs together, since inputs and outputs are connected at the lowest architectural level, independent of the configuration of the system. While certain configurations will produce oscillating outputs (for example, connecting an inverter's output to its input), these will not harm the hardware. This feature applies to any configuration error such as compilation errors or improper option settings, and thus will likely come in handy in classroom settings.

The Cell Matrix architecture also exposes new areas of research in dynamically modified circuits. This has at least two offshoots in EHW. First, you can experiment with the evolution of dynamic circuits, i.e., circuits that process not only binary data, but circuitry itself -- even their own circuitry, similar to MC Escher's "Drawing Hands." Second, if you are running a genetic algorithm (GA) on the hardware, you should be able to modify the GA itself while it is running, or have it modify itself, or have some other that is also onboard modify the GA. This leads to the possibility of higher-level evolution, i.e., form a population of GAs and evolve the evolutionary process itself.
The low level architectural support for dynamically-modified circuitry comes from the distributed, local, autonomous configuration mechanism possessed by every cell inside the matrix. We have used this *self-configurability* as a basis for much interesting research, and we see it as absolutely critical to making control systems for the extremely large systems that researchers are now working to achieve using nanotechnology.

Macias [REF P] used this distributed, local control to implement a GA in which the time it took to evolve from one generation to the next was independent of the population size. Sekanina [REF M] further demonstrated this $O(1)$ performance.

Cell Matrix hardware is a two dimensional or three dimensional grid of fine-grained programmable logic that permits run-time reconfiguration of regions of the hardware [REF U]. In fact, though, there is no distinction between “run-time” and “compile-time” on a Cell Matrix, since execution of circuitry on the Cell Matrix is used to further configure itself. Further distancing it from other types of programmable logic is the fact that reconfiguration can be (but need not be) organized and conducted as a local event, with cells reconfiguring neighboring cells. There is no hierarchy among the substructures in the hardware, no inherent specialization: they are all simply cells that can pass data to their neighbors and reprogram their neighbors.
Each individual cell can perform simple logic on the inputs it receives from its neighbors, and pass the results to its neighbors. Individual cells can also act as pieces of wire, passing signals from one cell to another, non-adjacent cell.

A four-sided cell consists primarily of a 16-row truth table, indicating the outputs to be generated to each of its four neighbors. A cell's “configuration” is simply the pattern of bits in its truth table. In addition to passing data, cells may pass truth table configurations, thereby allowing one cell to modify a neighboring cell's future behavior. From these fine-grained underpinnings, complex combinatorial circuits, state machines, memories, and other useful circuits are constructed. Many useful circuits (various kinds of flip-flops, 2-D memories, etc.) have been constructed for this architecture and are available as libraries within the Layout Editor, free software for creating Cell Matrix circuits [REF W]. To date, a complex, heterogeneous circuit containing a million cells has been successfully built and debugged using the online software tools and simulators. This circuit contains regions that are dynamically created while the circuit is operating, and was tiled over a large virtual Cell Matrix in simulation to create a self-diagnosing, self-repairing circuit [REF Q]. The simple hardware structure, combined with dynamic parallel configuration, fault checking, and fault handling, is expected to make the Cell Matrix architecture translate nicely to quantum dot, nanotechnology, and self-assembly manufacturing that is currently being studied and attempted [REF T].
References


-D:
http://www.cellmatrix.com/entryway/products/software/software.html


-W:
http://www.cellmatrix.com/entryway/products/software/layoutEditor.html

---

**Paper 4**

**Evolvable Hardware and Rapidly Reconfigurable Field-Programmable Gate Arrays**

John R. Koza

Stanford University
koza@stanford.edu
The dominant component of the computational burden of solving most nontrivial problems using techniques of genetic and evolutionary computation is the measurement of the fitness of each individual in each generation of the evolving population.

The commercial availability of rapidly reconfigurable field-programmable gate arrays (FPGAs) opens the possibility of exploiting the massive parallelism of such hardware by briefly *embodies* each individual of an evolving population *into hardware*.

One requirement for making this approach practical is that the FPGA be *infinitely reprogrammable* (i.e. the functionality of each function unit and the connectivity between the function units is stored in static random access memory).

A second requirement for making FPGAs practical for the inner loop of the genetic algorithm or genetic programming is that the time required for reconfiguration be measured in microseconds to milliseconds. For this purpose, reconfigurability includes the tasks of technology mapping, placement, routing, bit creation, and downloading.

Moreover, most commercially available FPGAs are vulnerable to damage caused by combinations of configuration bits that connect contending digital signals to the same line. The genetic algorithm starts with an initial population of randomly created individuals and uses
probabilistic operations to breed new candidate individuals. Thus, a third requirement is invulnerability to damage (or rapid determination of the acceptability of a particular combination of configuration bits).

The Xilinx XC6200 series of field-programmable gate arrays was the first commercially available chip with a publicly described architecture that offered infinite reconfigurability, rapid technology mapping, placement, routing, creation, and downloading of the configuration bits; and invulnerability to damage.

The Xilinx XC6216 chip contains a 64×64 two-dimensional array of identical cells. The functionality and local routing of each of the chip's 4,096 cells are controlled by 24 configuration bits per cell. Additional configuration bits are used to establish non-local interconnections between cells and the functionality of the input-output blocks located on the periphery of the chip. Each cell can directly receive inputs from its four neighbors (as well as certain more distant cells).

Statement of the Problem

The goal is to synthesize the design of a minimal sorting network for sorting seven items.

A sorting network is an algorithm for sorting items consisting of a sequence of comparison-exchange operations that are executed in a fixed order. Sorting networks are oblivious to their inputs in the sense that they
always perform the same fixed sequence of comparison-exchange operations, regardless of the inputs.

Sorting networks are of considerable practical importance. For example, are often part of the internal machinery of computers to sort internal variables for scheduling or channeling data.

In a sorting network, the two-argument **COMPARE-EXCHANGE** function changes the order of the to-be-sorted bits. The result of executing a \((\text{COMPARE-EXCHANGE } i \ j)\) is that the bit currently in position \(i\) of the vector is compared with the bit currently in position \(j\) of the vector. If the first bit is greater than the second bit, the two bits are exchanged. That is, the effect of executing a \((\text{COMPARE-EXCHANGE } i \ j)\) is that the two bits are sorted into non-decreasing order.

Table 1 shows the two results \((R_i \text{ and } R_j)\) produced by executing a single \((\text{COMPARE-EXCHANGE } i \ j)\) function. Note that column \(R_i\) is simply the Boolean AND function and column \(R_j\) is the Boolean OR function.

**Table 1 The **COMPARE-EXCHANGE** function**

<table>
<thead>
<tr>
<th>Two Arguments</th>
<th>Two Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_i)</td>
<td>(R_i)</td>
</tr>
<tr>
<td>(A_j)</td>
<td>(R_j)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
There has been a lively search over the years for smaller sorting networks (Knuth 1973) for various numbers of items. In U. S. patent 3,029,413, O'Connor and Nelson (1962) described sorting networks for 4, 5, 6, 7, and 8 items employing 5, 9, 12, 18, and 19 comparison-exchange operations, respectively. Later in the 1960s, Floyd and Knuth devised a 16-step 7-sorter and proved it to be a minimal 7-sorter. They also proved that the four other sorting networks in the 1962 O'Connor and Nelson patent were minimal.

Preparatory Steps for the Run of Genetic Programming

The initial terminal set for the result-producing branch, $T_{rpb-initial}$, for the problem of synthesizing the design of a sorting network for seven items is

$$T_{rpb-initial} = \{D_1, ..., D_7\}.$$

The initial function set for the result-producing branch, $F_{rpb-initial}$, is

$$F_{rpb-initial} = \{\text{COMPARE-EXCHANGE}, \text{NOP}, \text{PROG2}, \text{PROG3}, \text{PROG4}\}.$$

Here \text{NOP} is the zero-argument "No Operation" function.
The PROG2, PROG3, and PROG4 connective functions are versions of the PROGN function that sequentially evaluate their two, three, or four arguments, respectively.

Each program in the population has a uniform architecture consisting of one result-producing branch. Each individual in the population consists of a constrained syntactic structure composed of primitive functions from the function set, \( F_{\text{rpb-initial}} \), and terminals from the terminal set, \( T_{\text{rpb-initial}} \), such that the root of each program tree is a PROG2, PROG3, or PROG4; each argument to PROG2, PROG3, and PROG4 must be a NOP or a function from \( F \); and both arguments to every COMPARE–EXCHANGE function must be from \( F_{\text{rpb-initial}} \) (but not NOP).

Fig. 1 shows a program tree produced in accordance with this constrained syntactic structure. This program tree represents a minimal five-step sorting network for four items. As can be seen, the sorting network performs the five operations of (COMPARE–EXCHANGE 0 1), (COMPARE–EXCHANGE 2 3), (COMPARE–EXCHANGE 0 2), (COMPARE–EXCHANGE 1 3), and (COMPARE–EXCHANGE 1 2).

![Program Tree for a Minimal Sorting Network for Four Items](image-url)
Knuth (1973) has shown that if a sorting network correctly handles all $2^k$ fitness cases (consisting of all possible vectors of $k$ bits), it will correctly sort any $k$ numerical values. Thus, the fitness of each individual program in the population is based on the correctness of its sorting of the $2^k$ fitness cases. If, after an individual program is executed on a particular fitness case, all the 1's appear below all the 0's, the program is deemed to have correctly sorted that particular fitness case.

Since the goal is to evolve small (in fact, minimal) sorting networks, exchanges where $i=j$ and exchanges that are identical to the previous exchange are automatically edited out. Moreover, during the depth-first execution of a program tree, only the first $C_{max}$ COMPARE–EXCHANGE functions in a program are actually executed. $C_{max}$ is set to five more than the currently best known sorting network.

The fitness measure is multiobjective in that it involves both the correctness of the sorting network and parsimony (i.e., number of COMPARE–EXCHANGE functions in the network). Standardized fitness is defined in a lexical fashion to be the number of fitness cases (0 to $2^k$) for which the sort is performed incorrectly plus 0.01 times the number (1 to $C_{max}$) of COMPARE–EXCHANGE functions that are actually executed.

The population size was 1,000. The run is terminated when an evolved individual successfully handles all $2^7$ fitness cases.
Mapping the Fitness Evaluation Task onto the Xilinx XC6216 Chip

The problem of synthesizing the design of sorting networks was run on a host PC Pentium type computer with "HOT Works" PCI board (made by Virtual Computer Corporation) containing a Xilinx XC6216 field-programmable gate array.

In this arrangement, the host PC begins the run by creating the initial random population (with the XC6216 waiting). Then, for generation 0 (and each succeeding generation), the PC creates the necessary configuration bits to enable the XC6216 to measure the fitness of the first individual program in the population (with the XC6216 waiting). Thereafter, the XC6216 measures the fitness of one individual. Meanwhile, the PC simultaneously prepares the configuration bits for the next individual in the population. After the fitness of each individual in the current generation of the population is measured, the (relatively rapid) genetic operations (reproduction, crossover, and mutation) are performed on the population.

The clock rate at which a field-programmable gate array can be run on a problem is considerably slower than that of a contemporary serial microprocessor that might run a software version of the same problem. Thus, in order to advantageously use the Xilinx XC6216 field-programmable gate array, it is necessary to find a mapping of the fitness evaluation task onto the XC6216 that exploits at least some of the massive parallelism of the 4,096 cells of the XC6216.
Fig. 2 shows our placement on 32 horizontal rows and 64 vertical columns of the XC6216 chip of eight major computational elements (labeled A through H). Broadly, fitness cases are created in area B, are sorted in areas C, D, and E, and are evaluated in F and G. The overall arrangement of fig. 2 operates as an 87-stage pipeline (the 80 stages of areas C and E, the three stages of answer logic F, and four stages of padding at both ends of C and E). The figure does not show the ring of input-output blocks on the periphery of the chip that surround the $64 \times 64$ area of cells or the physical input-output pins that connect the chip to the outside. Two such $32 \times 64$ areas operate in parallel on the same chip.

![diagram](image)

**Fig. 2** Arrangement of major elements A through H on a $32 \times 64$ portion of the Xilinx XC6216 chip

For a $k$-sorter ($k \leq 16$), a 16-bit counter B (near the upper left corner of the chip in fig. 2) counts down from $2^k-2$ to 0 under control of control logic A (upper left corner). The vector of $k$ bits resident in counter B on a given time step
represents one fitness case of the sorting network problem. The vector of bits from counter B is fed into the first (leftmost) $16 \times 1$ vertical column of cells of the large $16 \times 40$ area C.

Each $16 \times 1$ vertical column of cells in C (and each cell in similar area E in fig. 2) corresponds to one COMPARE-EXCHANGE operation of an individual candidate sorting network. The vector of 16 bits produced by the 40th (rightmost) sorting step of area C then proceeds to area D. Area D is a U-turn area that channels the vector of 16 bits from the rightmost column of area C into the first (rightmost) column of the large $16 \times 40$ area E. The final output from area E in fig. 2 is received and analyzed by answer logic F for whether the individual candidate sorting network has correctly rearranged the original incoming vector of bits so that all the 0's are above all the 1's. The 16-bit accumulator G is incremented by one if the bits are correctly sorted. Note that the 16 bits of accumulator G are sufficient for tallying the number of correctly sorted fitness cases because the host computer starts counter B at $2^k-2$, thereby skipping the uninteresting fitness case consisting of all 1's (which cannot be incorrectly sorted by any network). The final value of raw fitness is reported in 16-bit register H after all the $2^k-2$ fitness cases have been processed.

The two large areas, C and E, together represent the individual candidate sorting network. The configuration of the logical function units and interconnection resources of the 1,280 cells in areas C and E become personalized to the
current individual candidate network in the evolving population.
For area C in fig. 2, each cell in a 16×1 vertical column is configured in one of three main ways. First, the logical function unit of exactly one of the 16 cells is configured as a two-argument Boolean AND function (corresponding to result $R_i$ of table 1). Second, the logical function unit of exactly one other cell is configured as a two-argument Boolean OR function (corresponding to result $R_j$ of table 1). Bits i and j become sorted into the correct order by virtue of the fact that the single AND cell in each 16×1 vertical column always appears above the single OR cell. Third, the logical function units of 14 of the 16 cells are configured as "pass through" cells that horizontally pass their input from one vertical column to the next. For area E, each cell in a 16×1 vertical column is similarly configured.

There are four subtypes each of AND and OR cells and four types of "pass through" cells. Half of these subtypes are required because all the cells in area E differ in chirality (handedness) from those in area C in that they receive their input from their right and deliver output to their left.

Within each cell of areas C and E of fig. 2, the one-bit output of the cell's logical function unit is stored in a flip-flop. The contents of the 16 flip-flops in one vertical column become the inputs to the next vertical column on the next time step.
Answer logic $F$ determines whether the 16 bits coming from the 80th column of the pipeline (from the left end of area $E$ of fig. 2) are properly sorted—that is, the bits are of the form $0i116-j$.

When the XC6216 begins operation for a particular individual sorting network, all the $16\times80$ flip-flops in areas $C$ and $E$ (as well as the flip-flops in three-stage answer logic $F$, the four insulative stages, and the "done bit" flip-flop) are initialized to zero. Thus, the first 87 output vectors received by the answer logic $F$ each consist of 16 0's. Since the answer logic $F$ treats a vector of 16 0's as incorrect, accumulator $G$ is not incremented for these first 87 vectors. A "past zero" flip-flop is set when counter $B$ counts down to 0 in fig. 2 As $B$ continues counting, it rolls over to $2^{16}-1 = 65,535$, and continues counting down. When counter $B$ reaches $2^{16}-87$ (with the "past zero" flip-flop being set), control logic $A$ stops further incrementation of accumulator $G$. The raw fitness from $G$ appears in reporting register $H$ and the "done bit" flip-flop is set to 1. The host computer polls this "done bit" to determine that the XC6216 has completed its fitness evaluation task for the current individual.

The above approach exploits the massive parallelism of the FPGA in six different ways.

First, the tasks performed by areas $A$, $B$, $C$, $D$, $E$, $F$, $G$, and $H$ perform disparate tasks in parallel in physically different areas of the FPGA.
Second, the two separate $32 \times 64$ areas operating in parallel on the chip perform identical tasks in parallel in two physically different parts of the FPGA.

Third, the XC6216 evaluates the $2^k$ fitness cases independently of the activity of the host PC Pentium type computer. This is yet another example (at the highest level) of performing disparate tasks in parallel.

Fourth, the Boolean AND functions and OR functions of each COMPARE–EXCHANGE operation are performed in parallel (in each of the vertical columns of areas C and E of fig. 2). This is an example of recasting a key operation (i.e., the COMPARE–EXCHANGE operation) as a bit-level operation so that the FPGA can be advantageously employed. It is also an example of performing two disparate operations (AND and OR) in parallel in physically different areas of the FPGA (i.e., different locations in the vertical columns of areas C and E).

Fifth, numerous operations are performed in parallel inside control logic A, counter B, answer logic F, accumulator G, and reporting register H. The parallelization inside answer logic F is especially advantageous because numerous sequential steps are required on a conventional serial microprocessor to determine whether $k$ bits are properly sorted. Answer logic F is an example of a multi-step task that has been efficiently parallelized and pipelined on the FPGA.
Sixth, most importantly, the 87-step pipeline (80 steps for areas C and E and 7 steps for answer logic F and accumulator G of fig. 2) enables 87 fitness cases to be processed in parallel in the pipeline.

Using the above approach, a speed-up of better than 46-to-1 was achieved.

Results

The design for a 100%-correct 16-step minimal 7-sorter (fig. 3) was successfully evolved in 69 minutes.

The evolved sorting network has two fewer steps than the sorting network described in the 1962 O'Connor and Nelson patent on sorting networks. The evolved 7-sorter has the same number of steps as the 7-sorter devised by Floyd and Knuth (Knuth 1973) subsequent to the issuance of the O'Connor and Nelson patent (and that has been proven to be minimal).

Fig. 3 Evolved 7-sorter with two fewer steps than the 7-sorter in the 1962 O'Connor and Nelson patent
Referring to the eight criteria in *Genetic Programming III: Darwinian Invention and Problem Solving* (Koza, Bennett, Andre, and Keane 1999) for establishing that an automatically created result is competitive with a human-produced result, this result satisfies the following criteria for human-competitiveness:

- The result was patented as an invention in the past, is an improvement over a patented invention, or would qualify today as a patentable new invention.
- The result is equal to or better than a result that was considered an achievement in its field at the time it was first discovered.

Additional details on work in this section can be found in Koza, Bennett Hutchings, Bade, Keane, and Andre 1997 and Koza, Bennett, Andre, and Keane 1999.

**References**


---

**Paper 5**

“On-Line Evolution of Spiking Neural-Network Topologies Using Partial Dynamic Reconfiguration on Virtex II FPGAs”

Andrés Upegui, Carlos Andrés Peña-Reyes, Eduardo Sánchez

Swiss Federal Institute of Technology, Logic Systems Laboratory, 1015 Lausanne, Switzerland.
(andres.upegui, carlos.pena, eduardo.sanchez)@epfl.ch, http://lslwww.epfl.ch

**Abstract**
There is no systematic way to define the optimal topology of an artificial neural network (ANN) for a given task. Heuristic methods, such as genetic algorithms, have been widely used to determine the number of neurons and the connectivity required for specific applications or families of them. However, artificial evolution uses to be highly time-consuming, making it sometimes unsuitable for online execution. Herein we present a digital hardware system that implements an online genetic algorithm that searches for the best topology to solve a certain task. Evolution is performed online thanks to the partial reconfiguration properties of Virtex II FPGAs. Different modules representing network layers are designed, and downloaded to the FPGA to find the best combination of layers. The genome maps the combination of these layers. The genetic algorithm execution time is reduced, since the fitness is computed on hardware and the downloaded reconfiguration streams have a reduced size.

1. Introduction

Artificial neural networks (ANNs) constitute the most common technique to solve problems with adaptivity requirements, the type of problems where the solution is not known in advance. Adaptivity is used in order to allow an ANN to perform a given task. Several types of adaptivity methods could be identified, according to the modification that is done. The most common methods modify either the synaptic weights [7] or/and the topology.
Synaptic weight modification is the most widely used approach, as it provides a relatively smooth search space. On the other hand, the sole topology modification provides a highly rugged landscape of the search space (i.e. a small change on the network results in very different performances), and although the adaptation technique explores the space of computational capabilities of the network in a substantial way, it is very difficult to find a solution.

A hybrid of both methods could achieve better performances, because the weight-adaptation method contributes to smooth the search space rendering easier to find a solution. Growing [4], pruning [4][8], and genetic algorithms [11] are widely used adaptive methods that modify the network topology that along with weight modification converge to a solution. We propose, thus, a hybrid method where an adaptation of the structure is done by modifying the network topology, allowing the exploration of different computational capabilities. The evaluation of these capabilities is done by weight-learning, finding in this way, a solution for the problem.

However, topology modification has a high computational cost. Besides the fact that weight learning can be time-consuming, in this case, it would be multiplied by the number of topologies that are going to be explored. Under these conditions, on-line applications would be unfeasible, unless there is available very good knowledge of the problem in order to restrict the search space just to tune certain small modifications on the topology.
A part of the problem can be solved with a hardware implementation: in this case the execution time is highly reduced since the evaluation of the network is performed with the neurons running in parallel, reducing the overall execution time of the system. However, a complexity problem remains: while on software, extra neurons and connections imply just some extra loops, on hardware implementations there is a maximum area (resources) that limits the number of neurons that can be placed on a network. This is due to the fact that each neuron has a physical existence that occupies a given area and each connection implies a physical cable that must connect two neurons. Moreover, if an exploration of topologies is done, the physical resources (connections and neurons) for the most complex possible networks must be allocated in advance, even if the final solution is less complex. This fact makes the connectionism a very important issue since a connection matrix for a high amount of neurons is considerably resource-consuming.

Xilinx FPGAs allow tackling this resource availability problem thanks to their dynamic partial reconfiguration (DPR) feature [3], which allows the reusing of internal logic resources. This feature permits to dynamically reconfigure the same physical logical units with different configurations, reducing the size of the hardware requirements, and optimizing the number of neurons and the connectivity resources.

In this paper we propose a method to use the DPR to solve the ANN topology-search problem. As search method, we
present a simple genetic algorithm [5], which is executed on-line. We present, also, a neuron model exhibiting a reduced connectionism schema and low hardware resources requirements.

2. Dynamic Partial Reconfiguration (DPR)

Field Programmable Gate Arrays (FPGAs) are devices that permit the implementation of digital systems, providing an array of logic components that can be configured in a desired way by a configuration bit-stream. This bit-stream is generated by a software tool, and usually contains the configuration information for all the internal components. Some FPGAs allow performing partial reconfiguration (PR), where a reduced bit-stream reconfigures just a given subset of internal components. DPR is done with the device active: certain areas of the device can be reconfigured while other areas remain operational and unaffected by the reprogramming [3]. For the Xilinx FPGA families Virtex, Virtex-E, Virtex-II, Spartan-II and Spartan-IIIE there are three documented styles to perform DPR: small bits manipulation (SBM), multi-column partial reconfiguration, independent designs (ID) and multi-column partial reconfiguration, communication between designs (CBD).

Under the SBM style the designer manually edit low-level changes. Using the FPGA Editor the designer can change the configuration of several components, such as: look-up-table equations, internal RAM contents, I/O standards, multiplexors, flip-flop initialization and reset values. However, it is recommended to not change any property or
value that could change routing, as it could produce internal contentions and damage the device. After editing the changes, a bit-stream containing only the differences between the before and after designs can be generated. For neural-network implementations SBM results inaccurate due to the low-level edition and the lack of automation on the generation of the bit-streams.

ID and CBD allow the designer to split the whole system in modules [2]. For each module, the designer must go from the HDL description through the synthesis, mapping, placement, and routing on the FPGA, independently of other modules. Placement and timing constraints are set for each module and for the whole system. Among these modules some are reconfigurable and others fixed. A complete initial bit-stream is generated for the fixed along with the initial reconfigurable modules. Partial bit-streams are generated for the reconfigurable modules.

The difference between these two styles of multi-column partial reconfiguration is that CBD allows the inter-connection of modules through a special bus macro, while ID does not. This bus macro guarantees that, each time partial reconfiguration is performed the routing channels between modules remain unchanged, avoiding contentions inside the FPGA and keeping correct connections between modules. While ID results limited for neural-network implementation, in the sense that it does not support communication among modules preventing any connectivity to be done, CBD is well suited for
implementing layered topologies of networks where each layer matches with a module.

CBD has some placement constraints, related to the dimension and position of the modules and the connectionism among them. Some of them are: (1) the size and the position of a module cannot be changed, (2) input-output blocks (IOBs) are exclusively accessible by contiguous modules, (3) reconfigurable modules can communicate only with neighbour modules, and it must be done through bus macros (Figure 1), and (4) no global signals are allowed (e.g., global reset), with the exception of clocks that use a different bit-stream and routing channels (more details on [3]).

![Figure 1. Design Layout with Two Reconfigurable Modules (figure from [3]).](image)

### 3. Neural Models
The inherent parallelism of neural network models has led to the design of a large variety of hardware implementations. Applications as image processing, speech synthesis and analysis, high energy physics, and so on, have found in neural hardware a field with promising solutions. When implemented on hardware, neural networks can take full advantage of their inherent parallelism and run orders of magnitude faster than software simulations, becoming thus, adequate for real-time applications.

The most common neuron models use continuous values as inputs and outputs, processed using logistic, gaussian or other continuous functions [4, 7]. Biological neurons do not behave in this way, but with firing rates: as a neuron receives input pulses by its dendrites, its membrane potential increases according to a post-synaptic response. When the membrane potential reaches a certain threshold value, the neuron fires, and generates an output pulse through the axon. The best known biological model is the Hodgkin and Huxley model (H&H) [9], which is based on the ion current activities through the neuron membrane.

However, the most biologically plausible models are not well suited for computational implementations. This is the reason why other different approaches are needed. The leaky integrate and fire (LI&F) model [1,10] is based on a current integrator, modeled as a resistance and a capacitor in parallel. Differential equations describe the voltage given by the capacitor charge, and when a certain voltage is reached the neuron fires. The spike response model order 0
(SRM₀) [1,10] offers a similar response to that of the LI&F model, with the difference that, in this case, the membrane potential is expressed in terms of kernel functions instead of differential equations.

Spiking-neuron models process discrete values representing the presence or absence of spikes; this fact allows a simple connectionism structure at the network level and a striking simplicity at the neuron level. However, implementing models like SRM₀ and LI&F on digital hardware is largely inefficient, wasting many hardware resources and exhibiting a large latency due to the implementation of kernels and numeric integrations. This is why a functional hardware-oriented model is necessary to achieve fast architectures at a reasonable chip area cost.

3.1. The Proposed Neuron Model

So as to exploit the coding capabilities of spiking neurons, our simplified model [12], as standard spiking models, uses the following five concepts: (1) membrane potential, (2) resting potential, (3) threshold potential, (4) postsynaptic response, and (5) after-spike response (see figure 2). A spike is represented by a pulse. The model is implemented as a Moore finite state machine. Two states, operational and refractory, are allowed (see figure 2).
During the operational state the membrane potential is increased (or decreased) each time a pulse is received by an excitatory (or inhibitory) synapse, and then it decreases (or increases) with a constant slope until the arrival to the resting value. If a pulse arrives when a previous postsynaptic potential is still active, its action is added to the previous one. When a firing condition is fulfilled (i.e., potential ≥ threshold) the neuron fires, the potential takes on a hyper-polarization value called after-spike potential and the neuron passes then to the refractory state.

After firing, the neuron enters in a refractory period in which it recovers from the after-spike potential to the resting potential. Two kinds of refractoriness are allowed: absolute and partial. Under absolute refractoriness, input spikes are ignored. Under partial refractoriness, the effect of input spikes is attenuated by a constant factor. The refractory state acts like a timer that determines the time needed by a neuron to recover from a firing; the time is
completed when the membrane potential reaches the resting potential, and the neuron comes back to the operational state.

Our model simplifies some features with respect to SRM₀ and LI&F, specifically: the post-synaptic and the after-spike responses. The dynamics of the system are also affected because of the way that several input spikes are processed: under the presence of 2 simultaneous input spikes, SRM₀ performs an addition of post-synaptic responses, while our model adds the synaptic weights to the membrane potential. Even though our model is less biologically plausible than SRM₀ and LI&F, it is still functionally adequate.

3.2 The Proposed Neuron on Hardware

The hardware implementation of our neuron model is illustrated in figure 3. The computing of a time slice (iteration) is given by a pulse at the input clk_div, and takes a certain number of clock cycles depending on the number of inputs to the neuron. The synaptic weights are stored on a memory, which is swept by a counter. Under the presence of an input spike, its respective weight is enabled to be added to the membrane potential. Likely, the decreasing and increasing slopes (for the post-synaptic and after-spike responses respectively) have been previously stored on the memory to be taken into account to compute the membrane potential.
Although the number of inputs to the neuron is parameterizable, increasing the number of inputs implies raising both: the area cost and the latency of the system. Indeed, the area cost highly depends on the memory size, which itself depends on the number of inputs to the neuron (e.g. the 16x9-neuron on figure 3 has a memory size of 16x9 bits, where the 16 positions correspond to up to 14 input weights and the pre-defined increasing and decreasing slopes; 9 bits is the arbitrarily chosen data-bus size). The time required to compute a time slice is equivalent to, at least, the number of inputs +1 (up to 14 inputs plus either the increasing or the decreasing slope).

We have also designed two other larger neurons: a 32x9 and a 64x9-neuron. Each one of them requires a larger memory and larger combinatorial logic to manage the inputs. Some synthesis results from the implementation of these neurons on a Virtex II-pro vc2vp4 FPGA are
illustrated on Table 1. The area requirement is very low compared to other more biologically-plausible implementations (e.g. Ros et al [6] use 7331 Virtex-E CLB slices for 2 neurons).

<table>
<thead>
<tr>
<th>Neuron Size</th>
<th>CLB Slices</th>
<th>Function Generators</th>
<th>Area Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td>16x9-neuron (14 inputs)</td>
<td>17 CLB slices</td>
<td>33 function generators</td>
<td>0.57% 0.55%</td>
</tr>
<tr>
<td>32x9-neuron (30 inputs)</td>
<td>23 CLB slices</td>
<td>45 function generators</td>
<td>0.75% 0.76%</td>
</tr>
<tr>
<td>64x9-neuron (62 inputs)</td>
<td>46 CLB slices</td>
<td>92 function generators</td>
<td>1.53% 1.53%</td>
</tr>
</tbody>
</table>

Table 1. Resource requirements for a single neuron on a Virtex II-pro vc2vp4.

4. On-line Evolution

Three main types of neural network evolutionary approaches might be identified: evolution of synaptic weights, evolution of learning rules, and evolution of topologies [11]. *Evolution of synaptic weights* (also known as learning by evolution) is a particularization of a parameter evolution, which is far more time-consuming than other learning algorithms, making it uninteresting for our implementation. *Evolution of learning rules* is a typical case of function approximation, where the learning rule is the function to be evolved, and the function that we want to approximate is the optimal learning rule (i.e. the fastest, the
simplest, the one who achieve the best generalization, etc). This issue could be of further interest for our approach but it is not the issue to discuss on this document. Evolution of topologies is the most interesting as it allows the exploration of a wider search space, and combined with weight learning it can become a very powerful problem solver.

In section 2 we discussed some of the features of DPR, concerning the layout of the achievable system. The result is a modular structure, where each module can communicate solely with his neighbour modules through a bus macro (Figure 1). This structure matches well with a layered neural-network topology, where each reconfigurable module represents a layer. Inputs and outputs of the full network should be previously fixed, as well as the number of layers, the connectivity among them (number and direction of connections), an input coding module, and an output performance-computing module. While each layer can have whatever kind of internal connectivity, connections among them are restricted to neighbour layers (forward or backward), allowing recursive connectionism.

For each module, there exists a pool of different possible configurations. A simple genetic algorithm [5] is the responsible for determining which configuration is going to be active. As illustrated in figure 4, each module can be configured with several different layer topologies, provided that they offer the same external view (i.e. the same inputs and outputs).
The genetic algorithm considers a full network as an individual. Several generic layer configurations are generated to obtain a library of layers, which can be used for different applications; however, for each application the genetic algorithm finds the combination of them that best solves the problem. Fixed modules contain the required logic to code and decode inputs and outputs and to evaluate the fitness of the individual depending on the application (the fitness could also be evaluated on the PC).

Given n modules, corresponding to an n-layer neural-network and c(i) possible configurations for the i-th module (i = 1,2,…n), the genome contains n genes, and each gene determines which configuration is downloaded to each module. The binary encoding length of the i-th gene is l(i) = log₂(c(i)). The length of the full genome is given by L=∑l(i). Figure 5 illustrates the structure of the genome, and the flow of the genetic algorithm.
5. Conclusions and future work

In this paper we present a method to evolve hardware neural networks on-line. Although the method has been conceived to work with spiking neuron models, it is applicable to any kind of digital neuronal hardware. Our long-term goal is to develop a tool including a library of generic neural layers, which allow the user to select the desired type of neuron, the number of layers, number of neurons-per-layer and the connectivity among them. Then, the user should define a fitness criterion in order to let the evolution finding the best network.

Spiking-neuron models seem to be the best choice for this kind of implementation, given their low requirements of hardware and connectivity, keeping good computational capabilities, compared to other neuron models [10]. Likewise, layered network topology, which is one of the most commonly used, seems to be the most suitable for our
implementation method. However, other topologies are still to be explored.

Different space search techniques could be applied with our method. Genetic algorithms constitute one of the most generic, simple, and well known of these techniques but we are convinced that it is not the best one. It does not take into account information that could be useful to optimize the network, such as the direction of the error. An example of this could be found on [4] where growing and pruning techniques are used to find the correct size of a network.

In a more general way, this work could be considered for any hardware structure with adaptivity requirements. For example fuzzy systems could be evolved in a structural way, finding the best fuzzification, inference, rules, and defuzzification stages, considering each one of them as a layer of the system, and creating libraries of parameterizable modules. On the same way it applies to processor architectures, signal processing, cellular automata, statistical methods, etc.

6. References


---

**Paper 6**

“From Evolutionary Design of Image Filters to Virtual Reconfigurable Circuits and Evolvable IP Cores”

**Lukas Sekanina**

Faculty of Information Technology, Brno University of Technology, Brno, Czech Republic.

[sekanina@fit.vutbr.cz](mailto:sekanina@fit.vutbr.cz), [http://www.fit.vutbr.cz/~sekanina](http://www.fit.vutbr.cz/~sekanina)

1. **Evolutionary Design of Image Operators**

Miller and Thomson have introduced *Cartesian Genetic Programming* (CGP) that exactly fits the needs of evolutionary design of small digital circuits nowadays [LS3]. In CGP, digital circuits are composed of programmable elements (gates) arranged in a regular pattern of $r$ rows and $c$ columns. A configuration bit stream determines the configuration of these elements, their interconnection and connection of primary inputs and outputs. Hence the CGP model corresponds to the structure
of chips of the Xilinx FPGA family. A number of innovative circuits - such as elegant multipliers - have been evolved from scratch [LS14]. However, these circuits are relatively small (e.g. a 3x3 multiplier). In order to evolve more complicated circuits, some domain knowledge has to be introduced. One of the approaches developed to design complex circuits has been known as the functional level evolution [LS4].

In order to evolve image filters which will be competitive with conventional filters, we have extended the CGP to the functional level [LS5]. In our case all inputs as well as outputs of programmable elements utilize exactly 8 bits and perform simple binary functions, for example: binary and, or, xor, addition, addition with saturation, average, minimum and maximum. Every image operator is considered as a digital circuit of nine 8bit inputs and a single 8bit output (the so-called 3x3 filter) which processes gray-scaled (8bits/pixel) images. Every pixel value of the filtered image is calculated using a corresponding pixel and its eight neighbors in the processed image. Three integers are needed to configure a single programmable element: two of them define the connection of element inputs and the third specifies its function. The design objective is to minimize the mean difference between the filtered image and the original image. We chose to measure the mean difference per pixel (mdpp), since it is easy for hardware implementation. The 256x256 pixel Lena image was utilized in the fitness function; the evolved filters were tested using other images. Similarly to conventional CGP, only a very simple variant of the evolutionary algorithm
has been developed: Only mutation of two randomly selected active programmable elements is applied per circuit. Four individuals with the highest fitness values are utilized as parents and their mutated versions build up the new population. We have tried to evolve innovative 3x3 image filters which can compete in terms of implementation costs and quality (mdpp) with conventional filters such as median filter, averaging using various coefficients and Sobel edge detector. First, we described these conventional filters in VHDL and synthesized them into Xilinx FPGA XC4028XLA. For instance, the median filter costs 4740 gates, averaging filters cost about 1400 gates and the standard Sobel detector requires 1988 gates.

In particular we evolved “salt and pepper” noise filters, random shot noise filters, Gaussian noise filters, uniform random noise filters, edge detectors and various application-specific image filters. The evolved circuits exhibit better quality than conventional circuits if mdpp is measured. It is interesting to observe how evolution discovered to employ available functions: While the shot noise filters are based on elements like minimum and maximum (i.e. the if-then-else suppression of corrupted pixels), Gaussian and uniform noise filters perform a sort of “averaging”. In some cases the evolved circuits require less of equivalent gates than the conventional circuits. It is mainly evident if an evolved circuit is compared with the median filter. For instance, the “salt and pepper” noise is traditionally suppressed by means of the median filter. However, we evolved RA3P5 and FETX0 filters that cost only 1702 and 2075 gates respectively. The FS3 and FS7
are edge detectors evolved (which cost 1350 and 2079 gates respectively) but they provide the similar quality to the Sobel operator. More than 500 image filters were evolved; the most interesting of them are presented and analyzed in detail in [LS9].

Furthermore, we were able to evolve all these filters with the additional requirement on testability. Note that testability is an important feature of circuits especially when the circuits have to be produced in large series or when we need to diagnose the circuits at a given working place (e.g. in space). We recognized that a filter is testable if (1) it is composed of elements that have the so-called I-mode of operation, (2) inputs of elements are not connected to the same data source and (3) synchronization registers are included. Then it is possible to test any element of the filter using the primary circuit inputs and outputs only, i.e. without any specialized datapath. Because we let the evolution produce only circuits of such the structure, the evolved filters have been not innovative only but also testable as seen in [LS7].

2. Virtual Reconfigurable Circuits

It takes one day to produce 160000 generations on a common personal computer (Pentium III/800 MHz), i.e. 3-10 hours to evolve a single filter. In order to speed up the evolutionary design and so possibly to apply the method in an adaptive embedded industrial image recognition system, we have to perform the fitness calculation in hardware. Although FPGAs are widely accessible reconfigurable
circuits, the following points represent main obstacles for their usage for evolvable hardware: (1) The reconfiguration system is not sufficient for evolvable hardware because the partial reconfiguration is not fully supported (the Xilinx XC6200 family is off the market now and the usage of JBits is too complicated) and FPGAs can only be reconfigured externally. (2) FPGAs are based on Configuration Logic Blocks (CLBs) whose granularity is too fine for direct evolution of complex circuits. And these complex circuits are required in real-world evolvable systems.

In order to overcome these problems, we have developed a new method that allows us to build a new reconfigurable platform using a common FPGA. This approach is known as the “virtual reconfigurable device” [LS9]. The idea is simple: a new reconfigurable circuit (including programmable elements, configuration memory and routing logic) is designed using resources available in an ordinary FPGA. The designer has only to prepare a configuration for the ordinary FPGA such that it creates a new virtual reconfigurable circuit. It means that the designer can construct a reconfigurable circuit that fits exactly the requirements of a given evolvable hardware-based application. For instance, one-clock partial reconfiguration can be realized if is needed in a given application. Similar approaches have been utilized in [LS1, LS10, LS12].

An implementation of the virtual reconfigurable circuit for the evolutionary design of image filters has been introduced in [LS8]. This circuit consists of 29 programmable
elements allocated in a grid of 8 columns and 4 rows. The last column contains only a single programmable element, which is always connected to circuit output. Every programmable element has two inputs and can be configured to perform one of 16 functions. Every programmable element is also equipped with a register to support pipeline processing. A column of four programmable elements is considered as a single stage of the pipeline. Hence the first filtered pixel is available at the output in the ninth clock; other pixels are available each in the single clock. A column of programmable elements is treated as a configuration unit of 48 bits. Therefore, the virtual reconfigurable circuit can be configured in eight clocks. However, its reconfiguration takes 1 clock, since reconfiguration as well as execution is pipelined and reconfiguration can effectively be overlapped by execution. The routing circuits are implemented using multiplexors. The 384-bit configuration memory is composed of Virtex slices; a slice containing two flip-flops is utilized to store two bits of the memory. All bits of the configuration memory are connected to multiplexors that control routing and selection of functions in programmable elements.

In order to find out how many Virtex slices are needed for the implementation of this virtual reconfigurable circuit we modeled the circuit by means of VHDL and synthesized the circuit into FPGA XC2V1000. This design costs 4879 slices (74357 equivalent gates) and it can operate at 134.8 MHz. Therefore, a single filter can be evaluated in 0.48 ms and full reconfiguration takes 7.4 ns during the
evolutionary design. We obtained a speeding up more than 70 in comparison with the software approach.

3. Evolvable IP Cores

Stoica et al. mentioned three years ago that “the path leads to the IP (Intellectual Property) level and evolvable hardware solutions will become an integrated component in a variety of systems that will thus have an evolvable feature” [LS11]. No other (implementation) details were given there. However, virtual reconfigurable circuits offer a way to implement such a component on an FPGA.

In fact the need for third party IP cores in FPGA designs is driven by increased size and performance of FPGA devices [LS2]. We can identify the following reasons for the usage of this sort of component approach: (1) reduction of time to market (because of reusability), (2) increase in reliability (because target systems are built from verified and tested components), (3) effective problem solving in the case of complex systems and (4) the component approach in fact determines the business model for producers. As an example, we can mention the processor MicroBlaze, offered as the IP core by Xilinx [LS13].

Evolvable IP cores can be defined by means of evolvable components that have been introduced in [LS9]. The evolvable IP core consists of virtual reconfigurable circuit, genetic unit and controller. Fitness calculation is performed outside the component, by an environment that has to send to the evolvable IP core the fitness value of any circuit that
has been uploaded into the virtual reconfigurable circuit. Since any evolvable IP will be available as a source code in HDL, the evolvable IP core can be reused in the same way as ordinary IP cores are reused. For instance, a vendor could sell an evolvable digital filter IP core instead of an “invariable” digital filter IP core.

Because the concept of HDL implementation of the virtual reconfigurable circuit has already been introduced in the previous section, we have to only describe the genetic unit and controller now. Generally, we have two options for their implementation: either to employ a general processor (such as the MicroBlaze) or to design a specialized circuit. The genetic unit has to only provide genetic operators, chromosome memory and fitness memory. The evolvable core communicates with the environment (that is represented by other components) by means of the controller. When the evaluation of a single circuit is finished then the fitness value is sent to the core. The core is to store the fitness value and to wait for another request from the environment. New configurations (and so whole populations) must be generated by the evolvable core autonomously. The evolvable core encapsulates the reconfiguration process that is invisible from the external environment. Because the fitness calculation is carried out outside the core, the core in principle supports dynamic fitness functions and open-ended evolution.

In order to show an example of the evolvable IP core we have developed in addition to the virtual reconfigurable circuit a simple genetic unit (including the controller) in
VHDL [LS6]. The unit is able to configure the virtual reconfigurable circuit and to communicate with the environment. The unit consists of chromosome memory (containing four 8x48-bit chromosomes), the best chromosome memory, the best fitness register, mutation unit, two control counters and the finite state machine that controls the unit and communicates with the environment. Because the virtual reconfigurable circuit has to be configured in 8 clocks per 48 bits, the organization of the chromosome memory is 4x8x48 bits and the best chromosome memory is 8x48 bits. The genetic unit was synthesized into various Virtex FPGAs. For instance, 2635 slices (38051 equivalent gates) are required for Virtex XC2V1000 [LS6]. The whole evolvable IP core has not been synthesized yet. However, it is possible to estimate that it will cost about 100k equivalent gates if a “home-made” genetic unit is implemented as a specialized circuit. And such a core can definitely be realized on an ordinary FPGA.

4. Summary

We have presented a class of digital circuits in which the evolutionary approach is a really successful design tool: evolution beats engineering approaches in terms of quality as well as implementation cost very often. Using the virtual reconfigurable circuit it is possible to implement the whole system on an ordinary FPGA and thus speed up the evolution substantially. Then an adaptive real-time evolvable hardware-based image filter can be realized for some industrial image recognition applications. However,
higher implementation costs are the main disadvantages of this method. Virtual reconfigurable circuits allowed us to think about evolvable IP cores. The proposed evolvable IP cores are reusable and tunable easily, since they are represented at the level of HDL source code. And so thinking toward the future, these evolvable IP cores could in fact determine business model for evolvable hardware.

5. Acknowledgement

Lukas Sekanina is supported from the Grant Agency of the Czech Republic under No. 102/03/P004 -- Evolvable hardware based applications design methods and the Research intention No. CEZ MSM 262200012 -- Research in information and control systems.

6. References


---

**Paper 7**

**Evolvable Hardware for Aerospace Signal Processing Applications**

Esther A. Hughes, Sharon S. Graves

NASA Langley Research Center
Hampton, VA 23681-2199
Sharon.S.Graves@nasa.gov

1. **Introduction**

Evolvable Hardware (EHW) is a novel field that meshes reconfigurable hardware and artificial intelligence to create autonomous systems that are physically reconfigurable, environment-aware and evolvable.

In recent years, the magnitude of distributed sensing and control requirements on aerospace vehicles has resulted in the need for increased reliability and signal integrity in signal processing hardware. Digital Signal Processing (DSP) has emerged as an enabling technology in a variety
of aerospace applications such as communications, video and audio compression, noise reduction, and so on. At NASA Langley Research Center, research under the Autonomous Robust Avionics (AuRA) program has focused on using Evolvable Hardware to extend reliability in demanding aerospace signal processing applications, where very high performance processing or very low power consumption is essential. Reliability issues range from fault-tolerant and degradation recovery, optimal signal processing and survivable NASA/DoD operating in extreme environments to intelligent adaptive and learning systems for protection of areas and security of communications.

The objective of this research is to develop a new generation of reliable hardware that is able to recover from faults as well as adaptively reconfigure to optimize signal processing and performance. Signal analysis plays a pivotal role in aero-acoustic measurements, data compression and visualization applications. Frequency domain analysis and spectral estimation are required for communications and real-time control of distributed sensing/actuator nodes. The optimization of bandwidth, range, precision and functionality are of paramount importance in these applications. Autonomous scaling decisions can significantly improve system performance, speed and accuracy. For example, functionality and speed of hardware can be optimized using fixed-point implementation compared to floating to select tradeoffs between range and precision. Often this optimization must
be done in situ and is dependent on the content of the input signals as well as environmental conditions.

Genetic Algorithms are applied to the active, automated synthesis of distributed services for building self-organizing sensor nodes that enable sensor devices to be self-aware, self-reconfigurable and autonomous. Sensor nodes dynamically adapt to device failure and degradation, optimizing communications bandwidth and speed for use in the interconnectivity of highly dense sensor nodes.

2. Background

The basic numerical operations in DSP algorithms are multiplications and additions. Multiply-add based DSP operations generally involve vector-matrix multiplications. Common examples of these DSP functions include the Fast Fourier Transform (FFT), the Discrete Cosine Transform (DCT), convolution, and digital filters (FIR, IIR, sampling rate converters). We propose to design, implement, and synthesize an innovative multiply-accumulate (MAC) unit that adapts its output data-width based on certain optimizing criteria. Interval methods will be used to implement the rounding hardware of the MAC unit in order to ensure precise and accurate results.

3. Interval Methods

The concept of interval analysis methods is to compute with intervals of real numbers in place of real numbers. In interval methods, intervals are used to represent a range of
numbers, as opposed to a single, exact number. An interval is a range of numbers, for example, the interval between 2 and 3 contains all the numbers between and including 2 and 3. The notation used is [2,3]. The interval [2,3] stands for the set of all possible numbers in that interval. Further, the width of an interval is a natural measure of the accuracy of an interval approximation. Performing arithmetic on intervals results in an interval answer that contains all the possible results. For example, [1,2] + [2,3] = [3,5]. After performing any interval computation, the resulting interval is guaranteed to contain the set of all possible answers, or for simplicity, the correct answer. Depending on the desired accuracy and hardware limitations, it is often difficult to represent floating-point numbers precisely using binary notation. Furthermore, floating-point arithmetic is affected by rounding errors, and can produce inaccurate results. However, interval arithmetic has the advantage of giving rigorous bounds for the exact solution. The potential drawback for using interval methods is that it takes twice the storage of ordinary arithmetic. As memory is cheap and continues to get less costly, that potential drawback becomes a non-issue.

4. MAC Architecture

The architecture for the MAC is shown below:
The Virtex-II family of chips will be used to implement the MAC unit. As shown in the figure, the unit takes two 16-bit inputs, and produces output results that range from 8 to 32 bits, in byte-sized increments. The architecture includes rounding hardware that is based upon rounding modes defined by the IEEE 754 standard for binary floating-point numbers. The rounding hardware is governed by optimizing controls used to find an optimal solution for the targeted DSP algorithm.

5. Rounding Hardware

The rounding hardware for the MAC is adaptive and responds to an 8-bit input variable that describes optimizing parameters. These parameters describe the following: which of the four rounding modes is to be used in determining the output result (round to zero, round to nearest, round up, round down), output data resolution (8 bits, 16 bits, 24 bits, 32 bits), fixed versus floating point implementation, and fitness controls (power, speed, accuracy, precision).

A block diagram of the rounding hardware for the MAC follows:
The hardware is adaptive in that depending on which rounding method is chosen, the datapath is altered accordingly. The result is then output to the output register and subsequently processed to determine if it is optimal or not.

6. Evolvable Hardware System

The MAC unit described above will be used as a core component in the realization of an evolvable hardware system. In the example below, a DSP algorithm is built around the reconfigurable MAC unit.

Genetic Algorithms (GA) will be used to produce the genotype that determines the configuration of the MAC unit. As shown in the figure above, we will start with a set of optimizing criteria that will be input into the GA. The GA will then produce a genotype that will be fed to the DSP algorithm along with the input signal to be processed.
A fitness test (based on a set of specific fitness controls) will be performed on the output result. If the output is determined to be optimal, then we are done. Otherwise, the process continues until an optimal solution has been reached.

This work will lead to the design of intelligent sensor/actuator nodes that are self-aware, self-reconfigurable, and autonomous. As a large part of sensing involves DSP algorithms, the MAC unit will be a natural building block for intelligent node design and for the inevitable networking of such intelligent nodes. We will begin with fixed-point implementation and then proceed to a floating-point implementation of the MAC unit.

---

**Paper 8**

“Evolvable Hardware and Biological Development”

**Tim Gordon**

Department of Computer Science, University College, London, WC1E 6BT, UK.

[t.gordon@cs.ucl.ac.uk](mailto:t.gordon@cs.ucl.ac.uk),

[http://www.cs.ucl.ac.uk/staff/T.Gordon](http://www.cs.ucl.ac.uk/staff/T.Gordon)

1. **Introduction**

---
At UCL we have used Virtex as the basis of a general-purpose intrinsic EHW evaluation platform. Here we present details of the platform and summarize a range of evolutionary problems we have explored, including our current work focusing on how biological development aids scalability in EHW.

2. **Virtex as a General-Purpose EHW Platform**

It is our view that the most interesting work in the field of evolutionary circuit design focuses on the discovery of innovative circuits. We required a general-purpose circuit evaluation platform to base future work on, and because of our interest in design innovation we wished to provide as many avenues of innovation for evolution to explore as possible. Although evolution can find innovative circuits using traditional design abstractions [1,2] if we allow evolution to have control over design decisions at a lower level of abstraction, we allow it more opportunity to surprise us with interesting designs. (For example the most unusual circuits in the literature were found by relaxing as many design rules as possible [3]). To evolve at low abstraction requires either extrinsic evaluation using detailed simulation of the physical processes of the hardware, or intrinsic evaluation using reconfigurable hardware. As a great deal of computing power is needed to run high numbers of low-level circuit simulations, we took the practical option of an intrinsic platform.
a) Selection of Xilinx’s Virtex

We selected Xilinx’s Virtex as our intrinsic circuit evaluation device on the basis of a number of criteria that Thompson listed in [4]. The most important of these was that Virtex devices could be reconfigured indefinitely and quickly (extremely quickly through partial configuration), that it had fast I/O for testing fitness cases, and that it was the first of a family of devices based on an architecture that would be commercially available in mass volumes and at low cost for some years; thus the results of experiments, even if tied to the architecture, would be of interest for many years. But perhaps the most important point was that of fine-grained reconfigurability. The finer the level of reconfigurability an FPGA provides, the lower the abstraction that evolution is able to work at.

The Virtex architecture is course grained, and complete details of the reconfiguration bit-stream, thus the finest details of the architecture, have not been disclosed by Xilinx. However Xilinx provides a Java API, JBits [5] that allows a fine-grained level of reconfiguration. The resources made available through the JBits API allow evolution to work at a very low level of abstraction beyond traditional design abstractions. Thus the only problem with the Virtex architecture is contention: a wire driven from two sources can cause a short circuit that can damage the device. We avoided contention by imposing constraints on the routing evolution was allowed to explore. Details of the constraints are given in our experimental descriptions.
b. **Platform Design**

Although circuits were evaluated intrinsically, the evolutionary algorithm driving the system was in software. To allow fast reconfiguration of the Virtex by a host PC we used a PCI-based XRC development board [6] hosting a XCV400 Virtex, reconfigured using the selectMAP interface.

JBits operates by modifying a file containing a pre-existing configuration bit-stream. For our platform this bit-stream contains a template circuit that provides a bus interface so that test data can be written to and read from the Virtex, and I/O buffers to pass test and result data to and from the area of the device that is evolved. When a circuit design is to be evaluated, JBits adds the circuit to the template design, and the file is written to the Virtex. The platform design is shown in Figure 1.
3. Low Abstraction Evolution

Our first experiments using Virtex served to demonstrate that useful circuits could be found by evolution in the design space of the Virtex architecture at very low design abstraction. Earlier work using Virtex for evolutionary circuit design had restricted the search either to very simple combinational circuits [7] or explored only lookup table (LUT) and flip-flop configurations, ignoring all the remaining logic and routing available on the device [8,9]. We wished to show that such strong restrictions were not always necessary.

3.1 Genotype Abstraction

To do this we allowed evolution to control as many resources as possible found in a generic CLB. The lowest level of access to a CLB that the JBits API provides is at the level of abstraction specified in the com.xilinx.JBits.Virtex.Bits package. A number of resources specified by the package were not found in all CLBs or were for routing over areas larger than the area of the FPGA we evolved, and so were not used. Also we had to prevent contention as the Virtex architecture allows two CLBs to drive the same routing wire. Hence we restricted the routing between CLBs by evolving only the PIPs that route the output bus to the single lines running between neighboring CLBs. Additionally eight of these were not
evolved as in some configurations they could still result in contention. All remaining resources in the JBits Bits package were available to evolution and are shown in Table 1.

In our chromosome, an integer gene represented each resource in each CLB that was to be evolved. The alleles that each gene could take on varied depending on the values the resource could take on in the JBits API. The only features that are not encoded in this way are the LUTs. Each LUT function was represented in the chromosome by 16 bits, which define its truth. All features that we evolved in a CLB and the number of alleles that the corresponding gene could take on are detailed in Table 1.

<table>
<thead>
<tr>
<th>I/O Type</th>
<th>Input Name</th>
<th>Alleles</th>
<th>Logic Type</th>
<th>Component Name</th>
<th>Alleles</th>
<th>Routing Type</th>
<th>PIP Details</th>
<th>Alleles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slice x LUT Input</td>
<td>SxF1</td>
<td>27</td>
<td>LUT Truth Table</td>
<td>SLICEx_F (x16)</td>
<td>16x2</td>
<td>Output MUX to Single PIP</td>
<td>OUT0_TO_SINGLE_EAST2</td>
<td>2</td>
</tr>
<tr>
<td>Slice x LUT Input</td>
<td>SxF2</td>
<td>27</td>
<td>LUT Truth Table</td>
<td>SLICEx_G (x16)</td>
<td>16x2</td>
<td>&quot;&quot;</td>
<td>OUT0_TO_SINGLE_NORTH1</td>
<td>2</td>
</tr>
<tr>
<td>Slice x LUT Input</td>
<td>SxF3</td>
<td>27</td>
<td>CLB Internal</td>
<td>YB.YB</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT0_TO_SINGLE_SOUTH3</td>
<td>2</td>
</tr>
<tr>
<td>Slice x LUT Input</td>
<td>SxF4</td>
<td>27</td>
<td>CLB Internal</td>
<td>YCarrySelect</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT0_TO_SINGLE_WEST7</td>
<td>2</td>
</tr>
<tr>
<td>Slice x LUT Input</td>
<td>SxG1</td>
<td>27</td>
<td>CLB Internal</td>
<td>Y.Y</td>
<td>3</td>
<td>&quot;&quot;</td>
<td>OUT1_TO_SINGLE_EAST3</td>
<td>2</td>
</tr>
<tr>
<td>Slice x LUT Input</td>
<td>SxG2</td>
<td>27</td>
<td>CLB Internal</td>
<td>AndMux</td>
<td>4</td>
<td>&quot;&quot;</td>
<td>OUT1_TO_SINGLE_NORTH2</td>
<td>2</td>
</tr>
<tr>
<td>Slice x LUT Input</td>
<td>SxG3</td>
<td>27</td>
<td>CLB Internal</td>
<td>ClockInvert</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT1_TO_SINGLE_SOUTH0</td>
<td>2</td>
</tr>
<tr>
<td>Slice x LUT Input</td>
<td>SxG4</td>
<td>27</td>
<td>CLB Internal</td>
<td>CInvert</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT1_TO_SINGLE_WEST4</td>
<td>2</td>
</tr>
<tr>
<td>Slice x Flipflop Input</td>
<td>SxCk</td>
<td>10</td>
<td>CLB Internal</td>
<td>YDin.YDin</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT1_TO_SINGLE_WEST5</td>
<td>2</td>
</tr>
<tr>
<td>Slice x Flipflop Input</td>
<td>SxSR</td>
<td>11</td>
<td>CLB Internal</td>
<td>YIfSetResetSelect</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT2_TO_SINGLE_EAST6</td>
<td>2</td>
</tr>
<tr>
<td>Slice x Flipflop Input</td>
<td>SxBX</td>
<td>17</td>
<td>CLB Internal</td>
<td>XIfSetResetSelect</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT2_TO_SINGLE_NORTH6</td>
<td>2</td>
</tr>
<tr>
<td>Slice x Flipflop Input</td>
<td>SxBY</td>
<td>17</td>
<td>CLB Internal</td>
<td>XCarrySelect</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT2_TO_SINGLE_NORTH8</td>
<td>2</td>
</tr>
<tr>
<td>Slice x Flipflop Input</td>
<td>SxCe</td>
<td>11</td>
<td>CLB Internal</td>
<td>X.X</td>
<td>3</td>
<td>&quot;&quot;</td>
<td>OUT2_TO_SINGLE_SOUTH5</td>
<td>2</td>
</tr>
<tr>
<td>Output Bus</td>
<td>OUT0</td>
<td>12</td>
<td>CLB Internal</td>
<td>XDin.XDin</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT2_TO_SINGLE_SOUTH7</td>
<td>2</td>
</tr>
<tr>
<td>Output Bus</td>
<td>OUT1</td>
<td>12</td>
<td>CLB</td>
<td>Cin.Cin</td>
<td>2</td>
<td>&quot;&quot;</td>
<td>OUT2_TO_SINGLE_WEST9</td>
<td>2</td>
</tr>
<tr>
<td>Output Bus</td>
<td>OUT2</td>
<td>12</td>
<td>CLB Internal</td>
<td>BxInvert</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT3_TO_SINGLE_EAST11</td>
<td>2</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>----</td>
<td>-------------</td>
<td>---------</td>
<td>---</td>
<td>-----</td>
<td>---------------------</td>
<td>---</td>
</tr>
<tr>
<td>Output Bus</td>
<td>OUT3</td>
<td>12</td>
<td>CLB Internal</td>
<td>Bylnvert</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT3_TO_SINGLE_EAST8</td>
<td>2</td>
</tr>
<tr>
<td>Output Bus</td>
<td>OUT4</td>
<td>12</td>
<td>CLB Internal</td>
<td>LatchMode</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT3_TO_SINGLE_NORTH9</td>
<td>2</td>
</tr>
<tr>
<td>Output Bus</td>
<td>OUT5</td>
<td>12</td>
<td>CLB Internal</td>
<td>Sync</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT3_TO_SINGLE_SOUTH10</td>
<td>2</td>
</tr>
<tr>
<td>Output Bus</td>
<td>OUT6</td>
<td>12</td>
<td>Slice x LUT Mode</td>
<td>DUAL_MODE</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT3_TO_SINGLE_WEST10</td>
<td>2</td>
</tr>
<tr>
<td>Output Bus</td>
<td>OUT7</td>
<td>12</td>
<td>Slice x LUT Mode</td>
<td>F_LUT_RAM</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT4_TO_SINGLE_EAST14</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Slice x LUT Mode</td>
<td>F_LUT_SHIFTER</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT4_TO_SINGLE_NORTH13</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Slice x LUT Mode</td>
<td>G_LUT_RAM</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT4_TO_SINGLE_SOUTH15</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Slice x LUT Mode</td>
<td>G_LUT_SHIFTER</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT4_TO_SINGLE_WEST19</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Slice x LUT Mode</td>
<td>LUT_MODE</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT5_TO_SINGLE_EAST15</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Slice x LUT Mode</td>
<td>RAM_32_X_1</td>
<td>2</td>
<td>&quot; &quot;</td>
<td>OUT5_TO_SINGLE_NORTH14</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1: Genotype Fragment for One CLB of the System.**
Where “Slice x” is Specified, Two Genes were Present, One for each Slice of a CLB

### 3.2 Evolving Adders

We used this representation to explore the evolution of a
two bit adder. This problem had already been studied intrinsically using Virtex by Hollingworth and Tyrrell [7], but with very restrictive constraints that limit the space searched by evolution to combinational configurations of LUTs.

The template circuit used for our exploration of the two bit adder with carry problem was as shown in Figure 1. In this case the input buffer consisted of five input registers to provide the circuit inputs (A0, B0, A1, B1 and Carry In) and the output buffer consisted of three output registers (Sum0, Sum1 and Carry Out). We set the evolved area of the Virtex to a 2x2 CLB array, which led to a chromosome of 604 genes. Fitness was calculated testing the circuit with all 32 truth table cases for the adder problem and summing the number of bits outputted correctly. Full details of the experimental setup can be found in [10]. Genetic parameters are shown in Table 2. 10 runs were performed with results shown in Table 3. Fitness and deviations have been scaled to 100.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>2 Member tournament</td>
<td>70%</td>
</tr>
<tr>
<td>Crossover</td>
<td>One-point</td>
<td>100%</td>
</tr>
<tr>
<td>Mutation</td>
<td>Point</td>
<td>5%</td>
</tr>
</tbody>
</table>

Other parameters: Generational GA with elitism, population size = 50, random initialisation.

Table 2: Parameters for the Evolutionary Experiments
Table 3: Results from 10 Runs of Intrinsic 2 bit Adder Evolution

<table>
<thead>
<tr>
<th>Mean Fitness of Best Solns</th>
<th>Std Dev of Best Solns</th>
<th>% Runs Perfect</th>
<th>Mean Gensto Find Perfect</th>
<th>Std Dev of Gensto Find Perfect</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.9</td>
<td>1.2</td>
<td>9</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5.8</td>
<td>6.6</td>
<td>6</td>
<td>1</td>
<td>9</td>
</tr>
</tbody>
</table>

3.3 Discussion

Our results showed that useful Virtex circuits could indeed be found by evolution working at a very low design abstraction. In addition we noted some interesting features of the circuits that we evolved. The Virtex CLB contains logic that is designed specifically to facilitate efficient arithmetic circuits. Many of our evolved adders made use of this carry logic, unlike the adders found by Hollingworth and Tyrrell [7], which were limited to searching a design space of combinational LUT logic only. We concluded that relaxing the design constraints facilitated the discovery of circuits that used the Virtex architecture more efficiently.

In [10] we compared our results with Hollingworth and Tyrrell’s and concluded that it was harder to find functional adders in the low abstraction space, but we noted that the huge increase in search space that resulted from allowing evolution to explore so much more of the Virtex architecture was not accompanied by a huge decrease in algorithm performance, rather only a moderate decrease. This suggested that not only could we find useful circuits in this low-abstraction Virtex design space, but also, for the
adder problem (and we suggest for similar arithmetic circuits) the low-level Virtex design space is more evolvable than a smaller yet more restrictive space that limits all logic to LUTs.

4. Using Development to Aid Scalability

Although we found circuits that performed useful computations, the circuits that we evolved were tiny by human design standards. Finding larger circuits using the system above would require unfeasibly large search spaces. However the key to scalability may be in modeling biological development.

4.1 Scalability whilst Preserving Innovative Search

Traditional evolutionary systems, including our work in section 3, evolve a genotype where each gene directly represents a single component in the phenotype. This is unlike natural systems. In nature, when a gene is activated it generates a protein. The activity of the gene is controlled by the presence of activators and inhibitors, which are just proteins that are in turn generated by other genes. This results in networks of gene regulation and autocatalytic protein generation called GRNs. GRNs can in turn regulate each other, thus forming successively more complex, modular sets of GRNs. At the terminals of these networks lie proteins that actually alter the phenotype structure. These are activated over time in complex patterns thus generating a complex phenotype. Thus the genome generates the organism from evolution’s smallest building
blocks, proteins, with the complexity of the organism resulting from layer upon layer of protein interaction. When GRNs are evolved, evolution is free to tinker at any level of abstraction – it can work at an abstract level by expressing a master control gene for a GRN, thus generating a whole module, or make a low-level change by altering a gene nearer the terminal of a GRN. This process is of great interest to us, as by modelling it we can bias evolution towards building large circuits from abstract modules without preventing it from sometimes innovating at lower levels of abstraction. Hence we designed a developmental system that maps to circuits on our EHW platform.

4.2 Modeling Protein Interaction

The exploratory developmental system was designed to model protein interaction so that auto-catalytic networks of proteins could form and is described in full in [11] and [12]. Proteins were modeled as binary variables, labeled A-E. Each gene was modeled as a rule. The precondition of the rule specified which proteins must be present (activators), and which must be absent (inhibitors) in the environment in order for a particular gene to activate. The post-condition of the rule defines the protein generated if the rule activates. Figure 2 shows an example rule. A set of rules makes up the chromosome and defines how the proteins interact over time. At a given time-step during development the environment is inspected to determine which proteins are present, then each rule is inspected to determine whether the environment matches the rule. If it
matches, the rule is activated and the protein defined in the rule’s post-condition is generated. This goes on to make up part of the protein environment of the following time-step.

<table>
<thead>
<tr>
<th>Protein:</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>Postcondition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>01</td>
<td>00</td>
<td>11</td>
<td>0111001</td>
<td></td>
</tr>
</tbody>
</table>

If **A present** and **D not present** and **E present** then **Generate protein B**

**Figure 2: The Developmental Rule of the Exploratory System**

The environment is segmented into a two-dimensional array of cells. This allows each cell to have a different environment from its neighbors, and so different rules can activate in each cell. Additionally cells can affect their neighbor’s environments at each time-step. In this initial model this was achieved by allowing a cell to inspect each of its Von Neumann-neighbor cells to determine what proteins they had just generated. If half or more of the cell’s neighbors had just generated a particular protein, then the cell considers the protein to be present in its own environment. Thus within each cell the rules activated (hence the proteins generated) are determined by the proteins generated at the last time-step in the cell and its neighbors. This results in auto-catalytic networks of protein interaction, which vary across the cellular array. The next section describes how we use this map of varying protein activity to alter the functionality of an array of Virtex
CLBs, thus generating a circuit.

4.3 Mapping Proteins to Virtex Circuit Designs

The developmental system described so far models the process of forming patterns of proteins – every rule post-condition that activates generates another protein. We needed a mechanism to map these patterns to a circuit. To achieve this, each cell in the developmental array mapped directly to a CLB on the Virtex. We then introduced a new set of rule post-conditions that would make a change to a component in a Virtex CLB (or its associated routing). These would serve as the terminal proteins that build up the circuit over time as discussed in section 4.1.

Over the course of development, the activity of each circuit-altering post-condition was recorded, and once development was complete the activities were compared to determine what alterations should be made to the template circuit. We used two methods to make this determination. The first was for components where we had to select a single state from several options, such as a LUT input, which can only be connected to one wire. These had a unique rule post-condition for every possible input wire, and the input wire that corresponded to the most active of these rule post-conditions over the course of development was connected in the final circuit. The other approach was for resources where several states could be active at once, such as the LUT truth table, where several p-terms can be set simultaneously to build up a logic function. Here we specified unique post-conditions for each p-term. If by the
end of development, post-condition activity had reached a predetermined threshold then the corresponding bit in the LUT truth table was set.

To give evolution control in this way over every component in a CLB as we have advocated, we would need at least one unique post-condition for every component in the CLB. Because the Virtex has a course-grained architecture with many components in a CLB, this would result in a very large space of component post-conditions that evolution would have to use to build up complex functions. Thus it seems that coarse-grained architectures like Virtex are not conducive to low-level development. This is in contrast to what we saw in section 3, where the Virtex architecture did not cause problems for component-based low-level evolution, as JBits gave us fine-grained control over the architecture.

We decided to initially work with a small set of components, and then explore the problems of using larger sets once we understood the dynamics of a simpler system. Thus we used only a single slice of each CLB, and introduced post-conditions to modify the logic and inputs of its two LUTs. Furthermore we restricted these inputs to eight hand-selected routes – signals from the two LUTs of each cell’s four Von Neumann neighbors. For more details of this mapping, see [11].

4.4 Experimental Results and Discussion

We evolved a genome of 80 initially random rules using the
same adder problem, genetic algorithm and parameters as used in section 3, except we increased the population size to 100. This was applied to a 5x2 CLB array on the Virtex. For comparison we also evolved a naive component-based mapping similar to that in section 2, but using only the components that the developmental system evolved. Results are shown in Table 4 with fitnesses scaled to 100.

<table>
<thead>
<tr>
<th>System</th>
<th>Mean Fitness of Best Solutions</th>
<th>Std. Dev. of Best Solutions</th>
<th>Best Solution Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developmental Evolution</td>
<td>81.9</td>
<td>2.63</td>
<td>84.4</td>
</tr>
<tr>
<td>Naive Evolution</td>
<td>95.4</td>
<td>4.10</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 4: Results from 5 runs of evolution with and without a developmental system

The results showed that the developmental system did not perform as well as the naive system. This might result from the increased epistasis of a developmental genotype-phenotype mapping.

However the developmental system demonstrated interesting results in the way it solved the problem. The solutions usually formed iterative patterns of reused structures, thus demonstrating one of the key advantages of development. Additionally, in [12] we presented a circuit evolved with this system where a handful of rules successfully encoded a combinational design abstraction by imposing a strong bias towards feed-forward circuits. We also showed that although development biased evolution towards this abstraction, evolution was capable of breaking the abstraction, and so was still free to explore at lower levels of abstraction if needed. This capability is precisely
what had motivated us to use development, as we explained in Section 4.1 Furthermore our model was inspired by the process of differentiation in biological development. In [10] we noted that solutions contained only a handful of distinctly different cell types – we had modeled a system that generated differentiated cell types as we had set out to do.

Although reducing the components available to evolution had limited the opportunities for innovation compared to the naive system used in section 3, it still left avenues for evolution to explore outside the scope of human design rules. In fact we found many examples where time-recurrent feedback loops had been used in the evolved circuits.

5. Improving the Developmental System

Although the system was capable of learning and representing design abstractions, it was not capable of solving the relatively simple the two-bit adder with carry problem, so needed further development. To identify the system’s shortcomings we considered how the problem might be solved in two stages. The first stage is pattern generation - we must be able to generate a pattern of proteins that is of sufficient complexity to map to a functioning circuit. The second stage is how this mapping takes place. Our more recent work has focused on the first stage - we have made a number of changes to our model of development in order to improve the computational power of pattern formation. These improvements involve
improving the intercellular communication, and are presented in [12], where we showed the changes greatly increased development’s ability to generate and maintain a range of patterns. We also showed that the computational power of the pattern formation model was now able to learn patterns that can successfully be mapped to functional adder circuits using hand-designed rules, using the terminal post-conditions discussed in section 4.3. We are currently exploring further changes to enhance the computational power of the developmental model. In the future we plan to also improve the second stage of the process by exploring a range of terminal post-conditions that still allow innovation through access to low abstraction design changes.

References


===================================================================

Paper 9

“THE e-LEM PROJECT”

Hardware Evolution of Neural Network
Circuit Modules for Brain Building

Hugo de Garis 1, Srivallaba Mysore 2, Yuya Sasaki 3, Ravichandra Sriram 1

1) Computer Science Dept, 2) Electronic & Computer Engineering Dept, 3) Department of Environment and Society, Department of Economics, Utah State University, Logan, UT 84322, USA
degaris@cs.usu.edu, ravichandras, srivallabam, slk1r@cc.usu.edu,

Abstract

This paper reports on ongoing work on the so-called “e-LEM” Project at our university. “e-LEM” stands for “electronic-LEM (Learnable Evolution Model)”, which is a proposed hardware implementation of a recent algorithm called LEM [2] that depending on the problem, can accelerate evolutionary computation (EC) by a factor of tens to thousands of times. By implementing this and similar algorithms directly in programmable hardware, it is hoped that neural network circuit modules will soon be evolvable in a about a second. This speed will make practical the evolution of large numbers of such modules (e.g. 10,000s and more) that can then be interconnected into artificial brain architectures.

1. Introduction

The primary research interest of the first author is the building of artificial brains. By this is meant the evolution (directly in electronics) of large numbers of neural network circuit modules, each module having its own function (and corresponding fitness
definition), and then interconnecting them to build artificial brains according to the architectures of human BAs (brain architects).

Modern electronics allows the above vision to be realized, thanks to the impact of Moore’s law. State of the art programmable chips contain up to ten million logic gates (e.g. Xilinx’s Virtex 2 chip family). With only a small number of such chips on a board, it should be possible to build quite elaborate artificial brains containing millions of artificial neurons contained in 10,000s of evolved neural network circuit modules.

However, the bottleneck in such an approach to building artificial brains is the time required to evolve the individual modules. For example, using an ordinary PC (Pentium 4), it took about 3 to 4 hours to evolve a 20 neuron neural network whose output signal followed closely a sinusoid target signal of a given frequency and amplitude. Obviously it will be totally impractical to build artificial brains with 10,000s (and higher orders of magnitude) of evolved neural network circuit modules, if the evolution is to be performed using ordinary software based techniques.

What is needed is a hardware-based approach to achieve the usual 100s to 1000s of times speed up of hardware-based solutions when compared with software-based solutions. Even better would be to combine the hardware approach with evolutionary computational algorithms that are a lot more efficient than the usual standard genetic algorithms.
Such algorithms now exist. Professor Ryszard Michalski of George Mason University, Virginia, USA, recently invented such an algorithm, called “LEM” [2]. This algorithm uses machine-learning techniques, and has been achieving speed-ups of hundreds of times compared to ordinary genetic algorithms, over many different application domains (although recent work in our group has shown that the degree of the speedup, if any, depends on the type of problem). The LEM algorithm and its major component, the AQ algorithm, are described in sections 2 and 3 respectively.

The basic idea of this paper and the e-LEM Project is to implement the LEM algorithm directly in programmable hardware. Since the hardware-software speedup is of the order of, let us say, 100s of times, and the LEM speedup can also be 100s of times faster than conventional genetic algorithms with their blind mutation and crossover operators, this would imply a total speedup factor of 10,000s of times, using a hardware based LEM approach compared to a software based approach on a PC. What took 3 hours on a PC with an ordinary genetic algorithm, would therefore take about a second on an e-LEM board.

This significant speed up makes brain building practical, since if it only takes a second to evolve a single module, 10,000s of them could be evolved in reasonable time.

The e-LEM Project will also be useful in the automation of multi-module evolution. Moore’s law will soon allow the creation of artificial brains with a billion artificial neurons, and hence probably millions of neural net modules. It will be totally impractical to have human brain-builder teams evolving
modules one at a time, defining each one’s fitness definition one after the other. Multi-module evolution will have to be automated. Our research team is thinking hard about how an e-LEM capability might be incorporated into algorithms specifically designed to automate multi-module evolution.

The e-LEM Project is not the first attempt at building artificial brains. The first author had built a 1st generation brain building machine, called the CBM [1] which was capable of evolving 3D cellular automata based neural network circuit modules in a few seconds. It could update the neural signalling of a 64000 module artificial brain in real time. Unfortunately, the previous lab of the first author went bankrupt in 2001 and the whole project died due largely to the fact that the hardware engineer of the project did not get fully paid by the lab.

A second generation brain building machine, called BM2 (Brain-building Machine, 2nd generation) is proposed, and is discussed in section 6 on future research.

The remaining contents of this paper are as follows. Section 2 introduces Prof. Michalski’s LEM algorithm that speeds up evolutionary computation by a factor of hundreds of times compared to blind genetic algorithms. Section 3 introduces the AQ algorithm that LEM uses. Section 4 describes an algorithm of the third author (the Sasaki algorithm) that is LEM-like, but operates bit-wise only (not over the whole bit-string), on the chromosomes used to evolve the modules. At the time of writing (March 2003) our team is still waiting to receive the source code of the LEM algorithm from Prof. Michalski’s team, modified to suit neural net module evolution. Prof Michalski’s research
assistant is still working on implementing it. Section 5 presents some of the (higher level) electronic design aspects of the implementation of the Sasaki algorithm, and its application to the evolution of a particular neural net model called “GenNet”. This model is also presented in this section. Section 6 discusses future research plans, and section 7 summarizes.

2. The LEM Algorithm

The LEM algorithm [2] is similar to a conventional (blind genetic operator based) evolutionary algorithm, with its fitness measurements, its survival of the fittest approach, its population of chromosomes, etc. It differs in that the usual genetic operators such as mutation and crossover are not used. In any given generation of the LEM algorithm, the fittest few chromosomes are said to belong to the “positive group” and the least fit are said to belong to the “negative group”.

A concept learning algorithm (called “AQ” [2], described in the next section) from the field of machine learning (one of Professor Michalski’s specialties) is then employed to generate a classification rule (usually in the form of disjuncts of conjuncts of attribute-variable pairs, e.g. IF ([haircolor = blue] & [height > 5’10”] OR [income > $1M] & [groupies > 2] THEN classification = male pop-star) which matches the symbolic descriptions of the +ve examples and none of the –ve examples.

A new generation of chromosomes is then generated from the classification rule, which match the +ves and none of the –ves.
The fitnesses are measured of the new generation and the loop is repeated.

Empirical results show that this machine learning approach, with its more intelligent guidance of the sampling of the possible chromosomes in the search space, can be *hundreds of times faster* than blind evolutionary-operator-based algorithms. This speed up was exciting to us due to its obvious implications for brain building.

3. The AQ Algorithm

The AQ algorithm [2] has been through many versions (in 2003, 20 versions). It is a concept-learning algorithm, that generates a classification rule (see the above section) that matches the symbolic descriptions of positive examples and does not match the descriptions of negative examples.

It works in the following way. Choose an arbitrary +ve example. Expand the allowed ranges of its variables (i.e. its attribute values) by maximally generalizing them, provided that its description does not match any –ve example. Then remove from the +ve examples any that match the expanded description. Now take another arbitrary +ve example and expand it maximally as before. Remove any other matching +ve examples from those remaining. Continue this process until no +ve examples are left. The resulting classification rule is the disjunct of the set of expanded descriptions. For details see [2] and the many papers on Professor Michalski’s web site [6].
4. The Sasaki Algorithm

While waiting for the LEM code to be developed, the 3rd author created a LEM-like algorithm which was 3 times faster than blind GAs on the same problem, so our research group chose to implement it in VHDL software simulation and programmable hardware. As mentioned in the previous section, once we have the details of the LEM algorithm for neural net evolution, with its potential of hundreds of times speedup, we will implement it in preference.

The Sasaki algorithm is similar to a conventional blind genetic-operator based evolutionary algorithm, with its fitness measurements, its survival of the fittest, its population of chromosomes etc. It differs in that the usual genetic operators such as mutation and crossover are not used. In any given generation of the algorithm, the fittest few chromosomes are said to belong to the “positive group” and the least fit are said to belong to the “negative group” (as in the LEM algorithm).

A new mutation algorithm (called “Softmax”), based on a Gibbs or Boltzmann probability distribution is used to calculate the probabilities of the state bits for the next generation of bit-string chromosomes. This works as follows –

A) The first step is to rank the chromosomes according to their fitness values.

B) Take the top \( n \) chromosomes as positive examples and the bottom \( n \) as negative examples. This part of the operation is identical to the LEM approach [2]. For each bit position in these
extreme examples, count the numbers of bit states (e.g. the number of zeros in a particular bit position in the $2n$ (+ve and -ve) bit-strings), by denoting the number of 0s or off states in the positive and negative examples as $N^0(pos)$ and $N^0(neg)$ respectively, and by denoting the number of 1s or on states in the positive and negative examples as $N^1(pos)$ and $N^1(neg)$ respectively.

Using the above 4 parameters, the probability that the “mutated” chromosome will have a 1 in a given bit position is calculated as follows.

\[
\Pr(1) = \frac{\varepsilon^{(N^1(pos) - N^1(neg))/n}}{\varepsilon^{(N^1(pos) - N^1(neg))/n} + \varepsilon^{(N^0(pos) - N^0(neg))/n}}
\]

where \(\Pr(1)\) is the probability in the next generation of chromosomes that a particular bit position will have a 1 or on state. \(\Pr(0)\) can be computed in the same manner, but it is simpler just to take the complement of \(\Pr(1)\), i.e. \(\Pr(0) = 1.0 - \Pr(1)\). The Gibbs or Boltzmann probability distribution uses the natural exponent as the base for each term. In this Sasaki algorithm, we use a parameter \(\varepsilon > 1.0\) instead of \(e\). This is useful, because by changing the value of \(\varepsilon\), we can influence the evolutionary speed or the evolvability.

Empirical results show that this “Softmax” mutation approach with its Gibbs or Boltzmann probability distribution approach of the sampling of the search space is \textbf{three to four times faster} than blind evolutionary-operator-based algorithms.
Our research group is still waiting to receive the LEM source code from the Michalski group, so in the mean time, we are using the Sasaki algorithm to accelerate our neural net module evolution speeds.

The first author teaches a PhD class in brain building, for which the speed of multi neural net module evolution is critical. Of course, LEM, with its probable hundreds of times speedup is preferable to the Sasaki’s 3-fold speedup, but at the time of writing, we only have the Sasaki algorithm available, so it is that algorithm that we are putting into (VHDL) hardware design. Once we receive the LEM algorithm, we plan to implement it too in programmable hardware. (See section 6 on Future Research).

5. Hardware Implementation of the Sasaki Algorithm Applied To Neural Network Evolution

This section describes the (high level) electronic design for the implementation of the Sasaki algorithm and its application to the evolution of a particular neural net model, called “GenNet”, whose characteristics are described next, followed by the principal electronic blocks needed for its implementation. The full VHDL (behavioral description level) code for this implementation is being written and is not included here. The principal hardware building blocks to implement the Sasaki algorithm as applied to the evolution of (GenNet model based) neural net modules, are as follows.
The Neuron Block, which receives weighted input signals from other neuron blocks. Its own output signal is fed to all other neuron blocks (i.e. a fully connected network) after multiplication by a weight value. The neuron block also receives an external input signal and a clock input.

The calculation of the output signal uses the following equations.

\[
\begin{align*}
\{ \\
Vi &= \sum WijNi + EXTi, \text{ where } Ni \text{ is the input from the } i \text{ th neuron, and } Wij \text{ is the weighting factor on the connection between neuron blocks } i \text{ and } j. \\
\text{Compute } F(Vi) &= \frac{2}{(1+\exp(-Vi))} - 1 \\
\}
\end{align*}
\]

At each clock edge, \( F(Vi) \) is the output value of neuron I, which is sent to all other neurons, as well as to itself. After a fixed number of clock ticks, the fitness function is calculated as follows -

\[
\text{Fitness } K = \frac{1}{\sum (F(Vp(t)) - T(t))^2} \text{ where } T(t) \text{ is the time } (t) \text{ dependent target function, and } Vp(t) \text{ is the output signal from the “output” neuron } p \text{ (of the network). Thus the highest fitness value is the reciprocal of the least mean square error i.e., the function that follows the target curve most closely.}
\]
The neuron blocks are interconnected and an arbitrary neuron \( p \) is chosen to be the output neuron.

For each neuron block \( i \), its signal outputs are the signal output values \( F(V_i) \). For the \( N \) neurons in the network, each neuron calculates its \( F(V_i(t)) \) values and outputs them to all other neurons.

The output neuron \( p \) (of the network) has additional circuitry to calculate the fitness of the network. See Fig. 1 (where \( i = p \)).

![Neuron Circuit](image)

**Figure 1: Neuron Circuit**

1. **Fitness Ranker Block**

   This block receives the chromosomes and their fitness values in the population, and ranks them.
3. **Sasaki Mutation Block:**

Following the Fitness Ranking, the Sasaki mutations are applied to the chromosomes, which takes place in the Sasaki Mutator block, as shown in Fig. 3. Since the chromosomes in the fitness ranker block are ranked in terms of their fitness values, the +ve and –ve samples are easily identified. Then each bit of the chromosomes is mutated based on the probability value calculated from the Sasaki formula.

This block therefore extracts the N1(pos), N0(pos), N1(neg) and N0(neg) values from the positive and negative examples. It passes these 4 values for each bit position through the logic to generates the probability values, and then takes the decision whether to flip the bit or not. The number of positive and
negative samples is a predetermined parameter so the number of address input values received is also fixed.

Figure 3: Circuit for Sasaki Mutation
Putting the above 3 components together gives a (quasi) complete block diagram. Certain handshakes and control signals are needed to complete the whole, that are used to control the proper sequential operation of the above set of blocks. Also, each block shown above contains sub blocks, which incorporate various arithmetic operations that make up the algorithm. We give only an outline of the circuitry design thinking in the above figures.

6. Future Research

The Sasaki algorithm and its application to the evolution of neural network circuit modules are currently being implemented in software simulation using VHDL, as presented in section 5. As soon as the Michalski team has finished the implementation of the neural net version of their LEM algorithm, our team at USU (Utah State University) will then simulate it too (in VHDL). We then intend taking these simulation results and using them to apply for research grants to implement them in real hardware (Xilinx’s Virtex 2 chips) on PCBs (printed circuit boards).

The second-generation brain-building machine called “BM2” (Brain-building Machine, 2nd generation) will be based on a hardware implementation of the LEM algorithm. This implementation will be a fundamental component of the BM2. This machine should easily outclass its 1st generation predecessor, the CBM [1], mentioned in the introduction. The BM2 is planned to evolve neural net modules in a fraction of a second and
then to store them in RAM. Human BAs (brain architects) will then specify the interconnections of these modules to build artificial brains. A similar approach was taken with the CBM.

The BM2, like its predecessor the CBM, will then be used to update the neural signalling of the interconnected artificial brain with its 10,000s of modules, and more.

7. Conclusions

The e-LEM Project aims to implement the LEM algorithm, (that can speed up evolutionary computation, depending on the problem, by hundreds of times due to its machine learning based approach) directly in programmable hardware. This will allow a neural net circuit module to be evolved in about a second. This speed will make brain building (i.e. the interconnection of 10,000s and more, of evolved neural net modules) practical. Once the e-LEM electronic board is implemented, it can be used as a vital component in the creation of a second-generation brain building machine, the BM2, to build artificial brains with up to a billion artificial neurons. Today’s electronics should make this possible. After all, the first-generation brain-building machine, the CBM[1] could update the neural signalling in real time of an artificial brain of 75 million neurons, and its chips (Xilinx XC 6264) dated from 1996. There have been several Moore doublings of electronic capacities since then.
References


[10] CS7910, “Brain Building” Class, PhD level, Utah State University, Power Point Notes, at http://www.cs.usu.edu/~degaris/lecture-notes.html, Click on “CS7910”