Ring Around the PIG: A Parallel GA with Only Local Interactions Coupled with a Self-Reconfigurable Hardware Platform to Implement an O(1) Evolutionary Cycle for Evolvable Hardware

From Proceedings of the 1999 Congress on Evolutionary Computation

Nicholas J. Macias
nmacias@cellmatrix.com

Abstract. The use of GAs in evolvable hardware is reviewed. A case is made for implementing as much of the GA in hardware as possible. The technical difficulties of using a standard GA with an FPGA are described. A new type of GA called a Ringed GA, which features only local interactions among individuals, is introduced. A new type of reconfigurable platform called the PIG is described. The use of the PIG to support local, parallel GA operations is explained. Experiments in evolving digital circuits using a ringed GA on the PIG are described. Conclusions and plans for future work are presented.

1 Introduction

The past few years have seen much progress in the application of genetic algorithms (GAs) [Koz92] to evolve digital circuits. There are many approaches to implementing such an algorithm. One is the all-software approach, with the evolved circuits being simulated in software, as in [Kit96]. Such an approach has the advantage of being able to run on an ordinary general purpose computer. However, since digital circuits generally operate in parallel, such serial simulations will be slower than the corresponding physical circuit would be, and as a result, each cycle of the GA can be prohibitively time consuming. A faster approach is to introduce a reconfigurable hardware device, such as an FPGA, and to cast each evolved circuit into hardware [Hem94]. This allows faster evaluation of an individual (digital circuit). Further speedups have been achieved by implementing other GA operations, such as mating, directly in hardware, leading to an all-hardware solution [Kaj98].

While an all-hardware approach shows significant improvement over an all-software or hybrid one, these solutions still tend to be fundamentally serial. The evaluation stage of the GA still involves evaluating each individual one at a time, which takes more time as the population size increases. Likewise for mating, which usually occurs one pair at a time. Hence a standard GA does not scale very well, as its execution time increases at least linearly with the population size.

An all-hardware GA can be made faster by evaluating all individuals simultaneously. This requires that all individuals in the population must exist simultaneously in the hardware, and that circuits for evaluating them must also be distributed throughout that hardware. This of course requires a sufficiently large hardware platform, which means the underlying hardware should be highly scalable. Since selection and mating are to occur in parallel across the population, these operations must be distributed as well. Since their execution time is to be independent of the population size, they should be based only on local interactions. Finally, since the output of mating is the creation of new digital circuits, the ability to reconfigure the hardware must also be distributed throughout the hardware itself. In other words, the hardware must be *self-reconfigurable*.

The present work introduces software and hardware to satisfy these requirements, which can be summarized as follows:

1. The underlying hardware must be scalable, to support large population sizes.
2. All GA operations must be based on local interactions only, to avoid increasing complexity as the population size grows.
3. The underlying hardware must be self-reconfigurable, since the outcome of mating (which is performed by the hardware itself) is the creation of a new individual in that same hardware platform.

Sections 2 and 3 describe a new type of GA, called a *Ringed GA* (RGA), which satisfies requirement 2. Section 4 describes a new type of reconfigurable device, the *Processing Integrated Grid* or PIG (US Patent #5,886,537), which satisfies requirements 1 and 3. Section 5 describes three experiments in evolving digital circuits using an RGA on a PIG. Section 6 states some conclusions and directions for future work.

2 Evolving Digital Circuits: The Basic GA Characteristics

The circuits to be evolved by the RGA are implemented on a reconfigurable array of interconnected cells, with each cell's configuration specified by a 16-row 4-column truth table. More details about these cells and their interaction will be described in Section 4. For the purpose of explaining the GA itself, the important fact is that each cell's configuration can be completely specified with 64 bits, representing the output values of the cells' truth table. By adopting an ordering of the cells making up a circuit, we can represent an n-cell circuit as a (64*n)-bit stream. These bitstreams are the chromosomes of our circuits, and it is these bitstreams which we mate and evolve in the GA.

There are several ways we can mate the bitstreams of two circuits. RANDOM mating picks out bits from each parent's chromosome, randomly selecting which parent each bit comes from. ROW mating selects entire rows from the truth table of each parent, thereby selecting bits in groups of four from one parent or the other. COLUMN mating is similar, but preserves columns of each parent's truth tables.

For the present work, ROW mating is the preferred style. This style reflects the notion that a circuit which works some of the time is actually operating correctly for certain input combinations, i.e., for certain rows of its truth tables.

For any mating style, we can introduce a bias to favor selection of bits from the parent with the higher fitness level. A mutation rate, which expresses the probability of a bit's value being toggled, can be set to help maintain diversity within the population. Further diversity is introduced through the generation of random members, which are created by generating random bitstreams. A bit probability can be set to control the likelihood of a bit's value being 1 or 0.

Scoring of a circuit's fitness is done differently depending on the circuit being evolved, but generally involves supplying test patterns to the circuits being tested, and comparing their outputs to the desired outputs. For simple combinatorial circuits, this might involve running the circuit through all possible input combinations and counting the number of times it produces the correct output. For sequential circuits, the circuit can be clocked and its outputs checked at each time step. For complex circuits where such exhaustive testing is prohibitive, it may be possible to carefully choose specific subsets of inputs to test, as is the case for evolving minimal sorting networks [Koz98]. Since the resulting score depends on how many input combinations are used, the scores are normalized to range between 0 and 1000, inclusive, with higher scores representing better fitness.

It must be stressed that this system evolves the actual configuration strings for the underlying hardware, as opposed to only modifying the way a fixed hardware configuration behaves. This is subtly different from, for example, the Firefly system, which uses a fixed FPGA configuration to implement a cellular automata (CA) machine, and then evolves rule tables for the CA. In M. Sipper's terms, Firefly uses two types of genomes. The organism's genome defines a cell's rule table, while the species' genome defines the underlying FPGA's configuration [Sip97]. These two genomes are fundamentally different in their meaning and use. In contrast, in the present work, the genomes of an evolving circuit are exactly the same type as the genomes used to implement the GA. There is no inherent hierarchy between the circuits being evolved and the circuits running the evolutionary algorithm. This suggests the possibility of a self-evolving system, in which the evolutionary algorithm itself undergoes modification.

3 The Ringed GA

A single evolutionary cycle of a traditional GA generally proceeds as follows:

- Evaluate the fitness of each individual
- Compare the scores of all individuals against each other
- Select a subset of the population based on those comparisons
- Mate certain pairs from this sub-population to form a new generation

The time for evaluating all individuals grows linearly with the population size, while comparing scores across the population has an order greater than O(n). This is because individuals are usually compared to all other members of the population to pick out the most fit members. Mating also grows linearly with the size of the sub-population.

Variations on a standard GA have been proposed to reduce the computational complexity of this cycle. One variation is the Island model, where evolution occurs in a set of sub-populations, with gradual migration among the sub-populations [Tom96]. Another is the Grid model, where individuals are distributed throughout a regular grid, and mating only occurs among nearby individuals. This locality of interaction is attractive for parallel interaction, but may lead to the development of semi-isolated niches [Tom].

The Ringed GA combines the best of both these models. Individuals are physically located throughout the space of the underlying hardware, and only physically-local interactions occur, but there is still a gradual migration of individuals throughout the population.

Figure 1 shows the basic setup of a small universe consisting of five rings. Each individual, which is actually a multi-cell circuit, occupies a single square. The individuals are arranged in concentric rings, with direct interaction occurring only among immediately adjacent individuals.

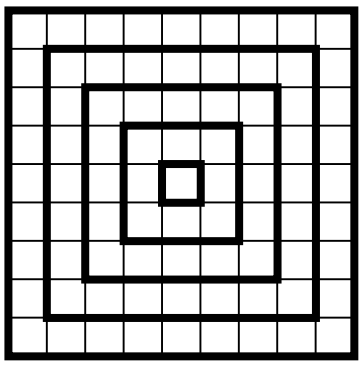


Figure 1. A 5-ring 81-Member Universe for the Ringed GA. Each box represents a single individual, containing a multi-cell digital circuit. Each circuit is compared with its two neighbors on the same ring.

The RGA begins with the creation of an initial population of random members, one per box in figure 1. This is done only once, as an initialization step at the start of the evolutionary process. Once this is completed, a series of evolutionary cycles are executed as follows:

1. The fitness of each member is computed. Note that this occurs in parallel among all individuals.
2. The fitness of each individual is compare to the fitness of its two immediate neighbors on the same ring. In figure 1, the fitness of individual **X** would be compared to that of individuals **Y** and **Z**. This also occurs in parallel across the entire population.
3. Each individual executes a mating step as follows:
   - If one of an individual's neighbors is more fit than itself, the individual mates with the most fit neighbor, and is replaced by the child.
   - If, however, the individual is more fit than either neighbor, that individual remains unchanged.

   There is one exception to these mating rules. If an individual is more fit than both its neighbors, and *significantly* more fit than the neighbor immediately counter-clockwise (meaning its fitness score is at least 100 points higher), it is copied verbatim to that less fit neighbor. This overrides any other mating operation for that less-fit neighbor. So if **X**'s score was at least 100 points higher than **Y**'s, **Y** would be replaced with an exact copy of **X**.

   Again, all mating occurs simultaneously across the population.

Steps 2 and 3, which collectively are called an *intra-ring cycle*, are repeated a fixed number of times. The basic idea is that adjacent individuals with similar fitness levels will generally mate, while extremely fit individuals will tend to propagate around the ring.

4. After a fixed number of these intra-ring cycles, an *inter-ring propagation* occurs, as shown in figure 2. This step begins with the generation of a new random member in the innermost ring, followed by the copying of certain members from inner rings to outer ones, as indicated by the arrows in figure 2. Again, this copying of individuals occurs entirely in parallel. The idea of this step is to migrate the results of intra-ring mating outward to the next larger ring. Diversity is introduced by the random member in the innermost ring, while the outer rings tend to be more homogeneous.

These steps are repeated until any individual achieves a perfect score, at which time that individual's circuit might be copied off to some other location. Alternatively, the evolutionary process could continue, to allow for continuous evolution, in case the fitness function is changing over time.

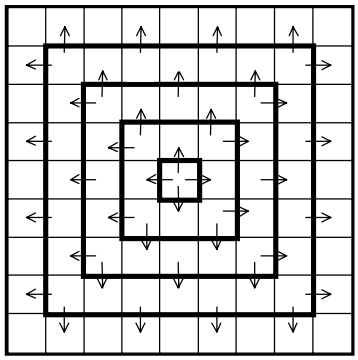


Figure 2. Inter-Ring Propagation Step of RGA. The central element is filled with a new random individual, which propagates to and replaces four spots on second ring. Corner cells of second ring propagate to 8 spots on third ring, and so on, as shown by arrows.

For the present work, evolution stopped as soon as a perfect score was detected.

While these steps are executed sequentially, each step takes a fixed amount of time, since all operations across the population occur in parallel. Each individual is responsible for scoring itself, based on a set of globally-supplied input and output data. Fitness comparisons occur only among immediately adjacent neighbors, and are negotiated by those neighbors themselves. Mating takes one of the following forms:

- Mate with clockwise neighbor,
- Mate with counterclockwise neighbor,
- Copy clockwise neighbor to yourself, or
- Do nothing.

In any case, each individual handles its own mating, and ultimately reprograms itself to take the form of the resulting offspring. The interring propagation also occurs between adjacent individuals, and is again handled by the individuals themselves. Therefore, all GA operations can occur in parallel across the population.

4 The Underlying Hardware: The PIG

Key to all of this work is the underlying hardware platform. This is a fundamentally new type of reconfigurable system, called the Processing Integrated Grid, or PIG (US Patent #5,886,537). The PIG is more than a reconfigurable device. It is a hardware platform for general reconfigurable work. Its key attributes are extreme parallelism, an infinitely scalable architecture, and the capacity for self-reconfiguration. In the context of implementing an RGA, the first two attributes help support a large population size, while self-reconfigurability allows the results of various operations to result in the reconfiguration of new circuits. This distributed reconfiguration control is essential for avoiding the bottlenecks inherent in serial, externally-controlled reconfigurables such as FPGAs.

The PIG is composed of a regular collection of simple, homogeneous processing elements called cells. Each cell has four sides, labeled N, S, W and E, as shown in figure 3. Each side has two inputs, C and D, and two outputs, also labeled C and D. We will ignore the C inputs for now, and assume they are always 0. Therefore a cell has four inputs (D_N, D_S, D_W and D_E) and 8 outputs (C_N, C_S, C_W, C_E, D_N, D_S, D_W and D_E), and thus its truth table has 16 rows and 8 columns.

Any such truth table can be implemented by a cell, and

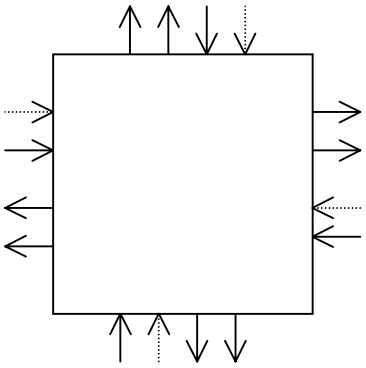


Figure 3. A Single PIG Cell. Each side has one input D and two outputs C and D. C inputs (dotted arrows) are assumed to be 0.

therefore a cell's configuration can be represented by the 128 output bits of its truth table. Figure 4 shows a sample cell, configured to realize the logical functions D_N=D_S, D_W=D_W and D_S=D_N AND D_E (all other outputs are 0). Table 1 shows the corresponding truth table for this configuration.

The PIG is organized as a regular two-dimensional array of these cells. Each cell is connected to four immediate neighbors, and exchanges two inputs and two outputs with each neighbor, as shown in figure 5. From this viewpoint,

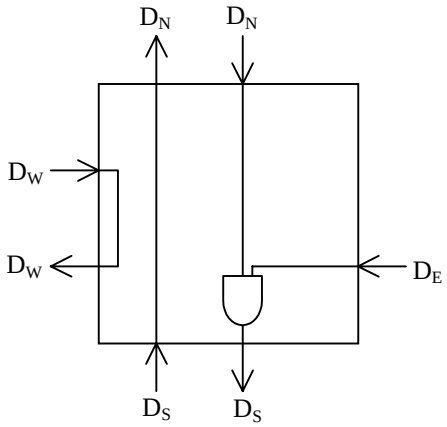


Figure 4. Sample Configuration of a PIG Cell corresponding to Table 1. This configuration implements the functions D_N=D_S, D_W=D_W and D_S=(D_E AND $D_{N)}$. All other outputs are 0.

the entire PIG is simply a collection of configurable combinatorial circuits. However, the C inputs have a special purpose. *If a C input is set to 1, the receiving cell is reconfigured based on its D inputs*. Thus any cell can configure any adjacent cell, and likewise can be configured by any of its immediately adjacent neighbors. Cells may pass around either plain data or reconfiguration information on the D lines. The interpretation of the passed bits is not based on the bits themselves, but rather on the C inputs to each cell. This interchangeability of code and data facilitates numerous dynamic operations, including the replication of cells, the dynamic construction of circuits, and the analysis of a cell's configuration. These capabilities lead to possibilities for such things as hardware libraries, virtual hardware, and fault-handling hardware.

INPUTS				OUTPUTS							
D_N	D_S	D_W	D_E	C_N	C_S	C_W	C_E	D_N	D_S	D_W	D_E
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	1	0
0	0	1	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0	0
0	1	0	1	0	0	0	0	1	0	0	0
0	1	1	0	0	0	0	0	1	0	1	0
0	1	1	1	0	0	0	0	1	0	1	0
1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	1	0	0
1	0	1	0	0	0	0	0	0	0	1	0
1	0	1	1	0	0	0	0	0	1	1	0
1	1	0	0	0	0	0	0	1	0	0	0
1	1	0	1	0	0	0	0	1	1	0	0
1	1	1	0	0	0	0	0	1	0	1	0
1	1	1	1	0	0	0	0	1	1	1	0

Table 1
Truth Table for Configuration shown in Figure 4

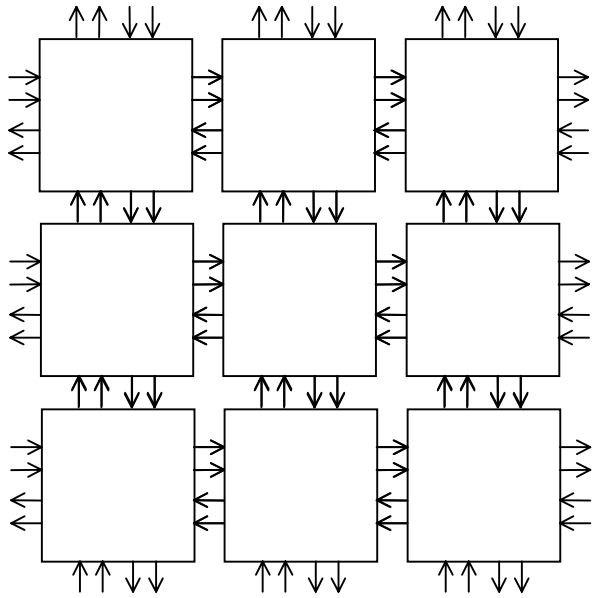


Figure 5. Sample 3x3 grid.

Note that each cell in the PIG is only connected to its four immediately adjacent neighbors. There is no direct connection from one cell to any non-adjacent ones. If you take two PIGs and attach them along an edge, you end up with a larger PIG, which functions exactly like the originals. You simply need to connect the C and D lines of the border cells, along with a pair of control lines (which are distributed to all cells identically). There are no address space issues or other impediments to scalability. Note that this has important manufacturing implications.

Since we are trying to evolve relatively simple, conventional circuits, we will assume that the C inputs and outputs are always 0, and each cell of a circuit is a simple combinatorial device. In this case, we only need to consider 64 bits of the truth table, corresponding to the values of the four D outputs. This is why the chromosome of each cell in a circuit is represented with only 64 bits. **However, by simply using a full 128 bits for each cell, we would be evolving circuits which are capable of reconfiguring other cells**. This ability to evolve circuits which themselves can create or modify other circuits reflects a fundamental property of the PIG. Such circuits can not be evolved, for example, on an FPGA, since the circuits which are configured inside the FPGA have no paths to the FPGA's configuration registers.

The individuals of the RGA are actually complex circuits, composed of multiple PIG cells. In general, there are two pieces to these individuals. There is the actual evolving circuit, which is what we usually talk about with respect to the RGA, and there is all the control circuitry needed to handle such things as mating, evaluation, etc. However, the entire individual is implemented on an array of identical PIG cells. No special hardware is used for any piece of the individual. The only difference from one piece to another is the configuration of the underlying cells. This is an important feature to keep in mind. Figure 6 shows the basic configuration of an individual. The figure only reflects an individual located along the top (North) of a ring, and not situated in a corner. For other individuals, the orientation of the I/O lines changes, but the internal circuitry is identical.

The Exploded Grid contains the actual circuit being evolved, along with additional circuitry which allows individual cell addressing. This grid is 9 times the size of the circuit it contains, e.g., a 4x4 circuit would have a 12x12 Exploded Grid. Recall that a cell is only directly connected to its four immediate neighbors, so in general, there is no way to access a non-adjacent cell. The additional circuitry within the Exploded Grid allows the circuit contained therein to function normally, but also allows direct access to each cell's configuration. The Row and Col inputs select a cell, current configuration data is read from the π_{out} line, and new configuration data is supplied via π_{in}. The inputs to the circuit under test are available on the In lines of the Exploded Grid, and the circuit's outputs appear on the Out lines. Thus the circuit within the Exploded Grid can be configured, tested, and reconfigured as necessary

Again, it is important to note that, as with all circuitry in figure 6, the Exploded Grid is not composed of any special hardware. Its underlying hardware is identical to all other hardware in the PIG. It is composed of a set of PIG cells, identical to all other cells in the PIG, except for their configuration data. Any special hardware requirements are met through the appropriate configuration of PIG cells. This is one of the PIG's most important features.

The remainder of the circuitry in figure 6 supports direct implementation of the RGA evolutionary cycle. The Row, Col, Cmd, Input and Des Out lines of all individuals are connected in parallel to a global controller. All other I/O lines are connected to adjacent individuals. The output of the evolving circuit is compared to the desired output, Des Out, and the Score Register is adjusted accordingly. This score is then compared to the scores of the individual's two neighbors, and a mating operation is chosen and recorded in the Mating Op Reg. This is then used to combine configuration information (truth tables) from the individual itself and its neighbors and produce configuration data for a child circuit, which is then loaded into the Exploded Grid. Interring propagation occurs similarly, with the π-Select Logic choosing configuration information from an inner ring where appropriate. Creation of a perfect individual is signaled by the Perf line, and that individual's configuration can be read from the π_{Perf} line.

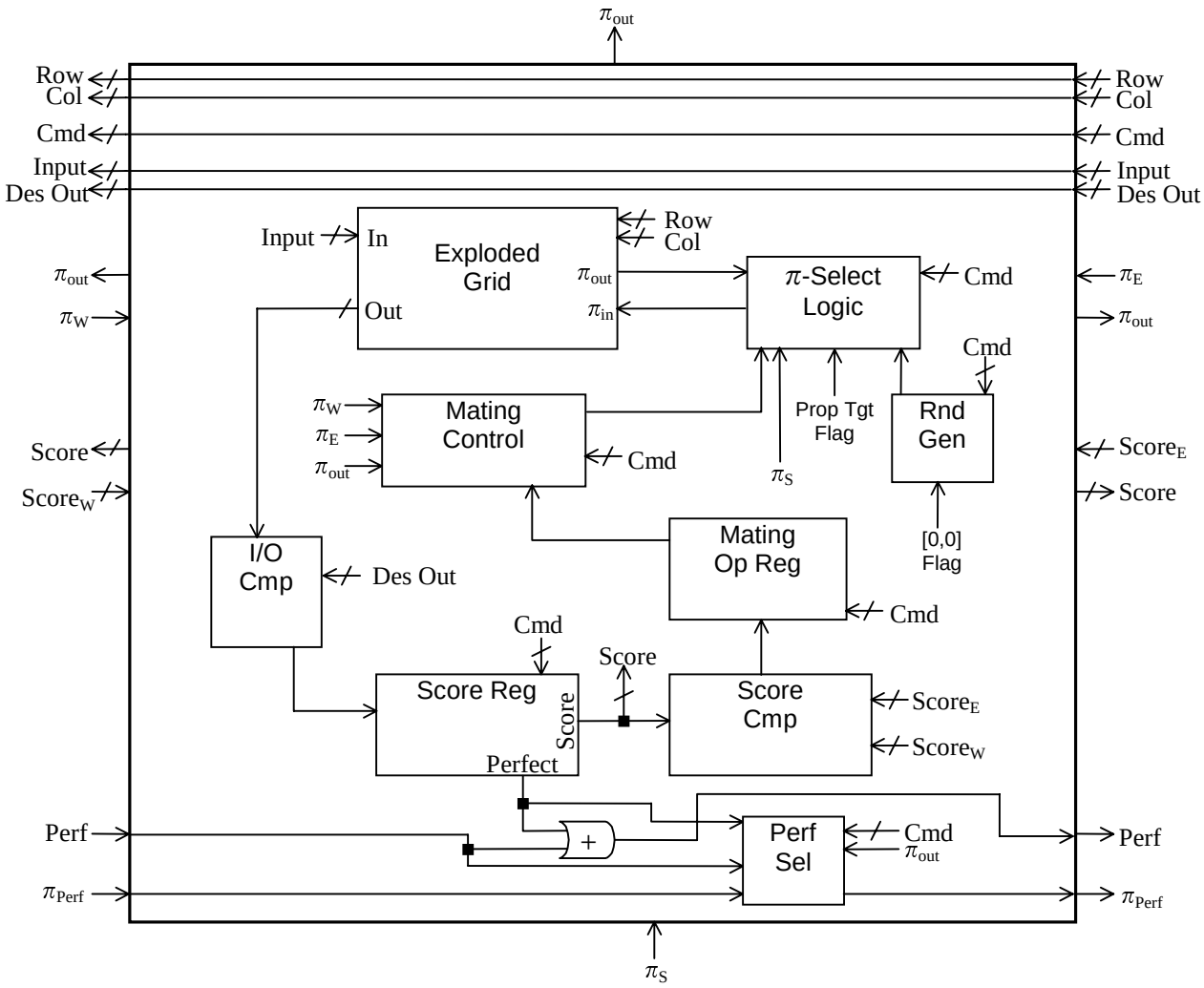


Figure 6. Single Individual for the RGA. Configuration shown and comments below are for individual along top of a ring and not in a corner. Actual evolving circuit is stored in the Exploded Grid. Row, Col, Cmd, Input and Des Out come from global controller, and are sent in parallel to all individuals. Cmd selects the RGA operation, Row and Col access individual PIG cells within the individual's circuit, Input is a test input, Des Out is the desired output from the circuit. All other I/O lines connect to immediately neighboring individuals. π lines carry configuration information. Perf indicates an individual along the row has achieved a perfect score. Configuration of leftmost perfect individual appears along π_{Perf}.

Figure 6 is only one possible implementation of an individual. Other than the Exploded Grid, the circuitry is just standard digital logic, and the global controller is a simple state machine. For example, the cells of the circuit within the Exploded Grid are accessed one at a time (but in parallel across all individuals). Other arrangements are possible, including having multiple π-lines to allow an entire NxM grid within the Exploded Grid to be reconfigured in parallel. This is a classic space-time tradeoff. For the price of more complex circuitry, you can achieve more parallelism.

With certain reconfigurable devices, there is a natural flow of information from one side of the device to the other, and thus circuits having unstable feedback paths are avoided. On the PIG, there is no such natural flow direction. This leads to the possibility of instabilities in a random circuit. Even for a simple 2-cell circuit, 12.5% of the possible configurations are unstable. While circuits can easily be built to detect and eliminate these instabilities, an easier solution is to constrain the truth table of each cell. We adopt a convention that for each cell, only the DS and DE outputs may be non-zero. Thus, information tends to flow from the NW corner to the SE corner of a rectangular circuit.

A prototype of the PIG has been built on a small scale. However, its practical use requires an extremely large number of cells, which current technology is unable to manufacture cost-effectively. Still, the simplicity of each cell and the regular interconnection scheme make the PIG an ideal candidate for exploiting emerging technologies such as nanotechnology [Dre86], biological computing [Lip96], and bistable quantum dots [Len93]. For the present work, a PIG

simulator was used for executing the individuals' circuits, while the GA operations themselves were implemented in C. It is hoped however that this work on the ringed GA, as well as work on other PIG applications, will provide incentive for the subsequent development of a large-scale physical PIG.

5 Experiments and Results

Experiments were performed to evolve three different digital circuits. In all cases, the experiment ended with the evolution of a circuit which performed the desired task perfectly. RGA parameters were chosen more or less at random (but within reason), and were kept constant across all three experiments. Specifically, when generating random individuals, a bit probability of 0.15 was used (meaning the probability of a 1 was 0.15). In mating operations, the probability that a bit comes from the parent with the higher fitness value was set to 0.60. The mutation probability (for each bit generated in the child) was set to 0.0125. The universe consisted of 8 rings (225 individuals), and an intra-ring cycle length of 9 was used. There doesn't seem to be anything magic about these particular settings, and successful evolution was observed with many other settings as well.

The first experiment was to evolve a four bit odd parity generator. Figure 7 shows the setup of the desired circuit. The goal was to evolve a circuit consisting of 16 PIG cells in a 4x4 grid, with four input bits, B_0-B_3, and one output bit, B_{out}, such that the number of ones among $\{B_0,B_1,B_2,B_3,B_{out}\}$ was odd. The test data consisted of all possible input combinations of four bits, tested in binary order. The fitness score was a simple count of the number of input combinations for which the correct parity was generated. Figure 8 shows the results of this experiment. As can be seen, a perfect circuit was evolved after 9 generations.

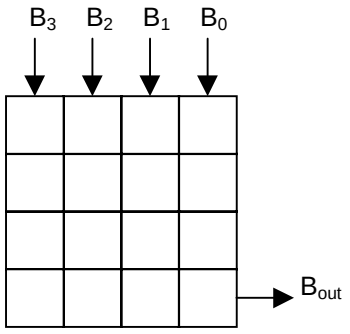


Figure 7. 4x4 parity generator template. Each square represents a single PIG cell. The goal was to configure all 16 cells so that B_{out} is the odd parity bit for B_0-B_3.

The second experiment was to evolve a 4-1 multiplexer. Figure 9 shows the setup of the desired circuit. The circuit accepts two select bits, S_1 and S_0, and four general inputs X_0-X_3. The circuit should select one of the inputs based on

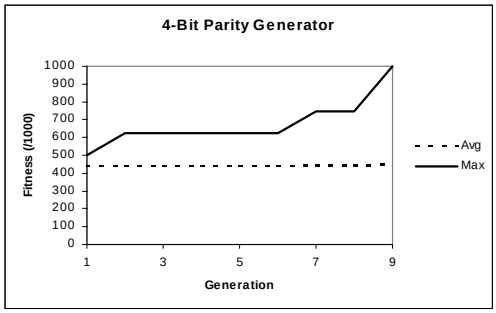


Figure 8. Results of parity generator evolution. A perfect odd parity generator was evolved in 17 cycles.

S_0 and S_1, and send that input to the output. The test data consisted of all possible input combinations of six bits (two select inputs and four X inputs), tested in binary order. The fitness score was a simple count of the number of input combinations for which the correct output was generated. Figure 10 shows the results of this experiment. In this case, a perfect circuit was evolved in 342 generations. This is an impressively low number of generations, considering that the search space for this problem consists of (4 cells)x(4 cells)x(4 rows)x(2 columns) = 128 bits, or approximately 10^{38} possible circuit configurations, *with the row constraint enabled* (without the row constraint, this number jumps to 2^{1024} or approximately 10^{308}).

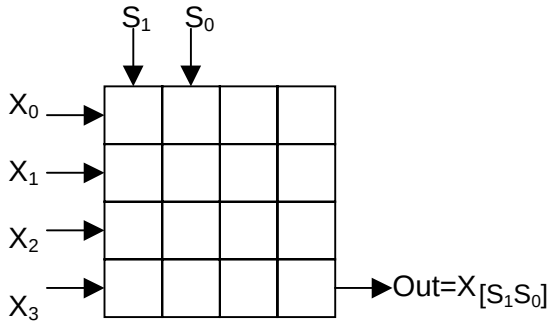


Figure 9. 4x4 template for 4-1 Multiplexer. The circuit being evolved should select one input from $\{X_0,X_1,X_2,X_3\}$ based on S_1 and S_0.

Since most complex circuits are sequential in nature, the RGA was used to generate a sequential circuit. The third experiment was to evolve a three-bit counter. Since sequential circuits require memory and feedback paths, a generic template for an 8-state sequential machine was built, as shown in figure 11. This circuit consists of three independently clocked D-type flip flops. All flip flop outputs (q_0-q_2) are sent to the left side of a blank 5x7 subcircuit, while all inputs to the flip flops (D_0-D_2 and CLK_0-CLK_2) are read from the bottom of that subcircuit. Additionally, the circuit accepts a single external clock signal, CLK_{in}. The subcircuit is a blank combinatorial circuit, which must be evolved to produce the correct sequential behavior of the entire circuit.

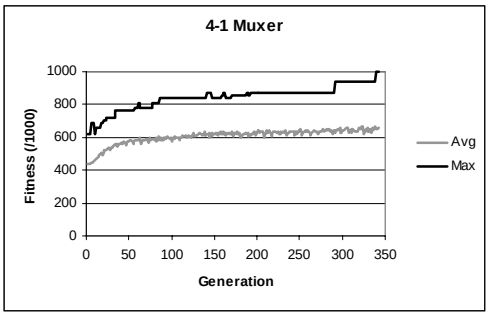


Figure 10. Results of multiplexer evolution. A perfect circuit was evolved after 342 evolutionary cycles.

Note that this is similar to the setup of some FPGAs [Act95], but that on the PIG, the flip flops and feedback paths are also implemented in PIG cells, as is the 5x7 subcircuit to be evolved.

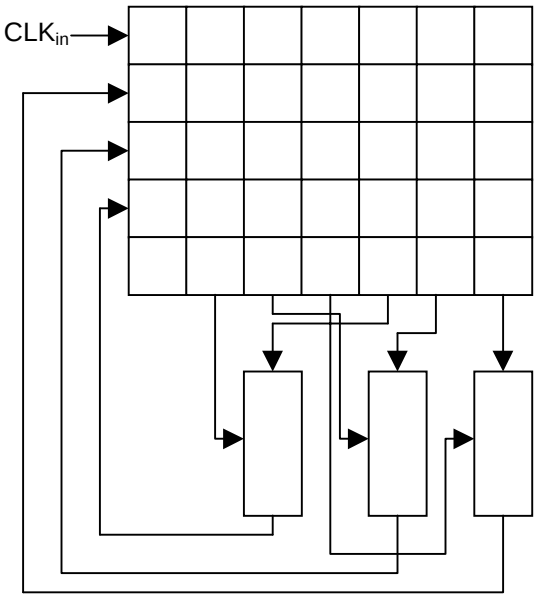
The desired behavior of this circuit was to accept a single external clock input, and produce sequential 3-bit integers on each falling edge of the external clock. The 3-bit output value was taken from the q outputs of the flip flops, labeled Q_0-Q_2 in figure 11. To evaluate the circuit, the clock input was toggled high and then low a total of 16 times, enough to cycle the output twice. The fitness scoring was a little more complex for this circuit than for the first two. As a counter, a simple NOP circuit (q_i=0) would score 125, without capturing any of the desired essence of the circuit. Therefore the circuit was scored, following each downward clock transition, as follows:

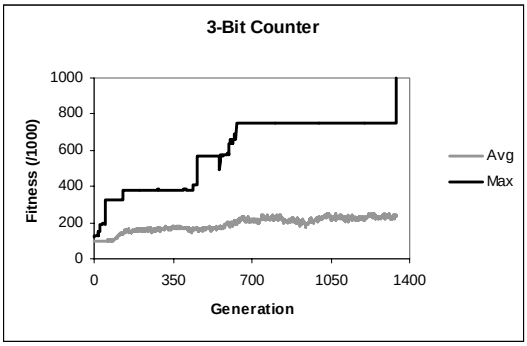
- If the output matches the desired output (next sequential number modulo 8), add 30 points to the score
- else if the output is one more than the previous output value (increment), add 5 points
- else if the output changes at all, add one point
- otherwise add no points for that clocking operation.

Figure 12 shows the results of this experiment. Here the system required 1339 generations before converging on a perfect solution. Again, with row constraints, the total search space for the 5x7 circuit is approximately 10^{84} possible circuit configurations.

6 Conclusions and Future Work

A new type of genetic algorithm, the RGA, has been described and shown to be both feasible and successful in the evolution of both combinatorial and sequential digital circuits. The RGA is inherently parallel, leading to an O(1) evolutionary cycle. This makes the RGA attractive for evolving large, complex circuits which may require a large population size. While it is not possible to implement a fully-parallel RGA on an FPGA, a new type of self-reconfigurable hardware device, the PIG, is well suited to implementing an RGA.





While other all-hardware parallel GAs have been implemented [Hig94, Kaj98], these generally involve custom hardware specifically designed for implementing the GA. In contrast, the PIG's hardware is in no way specialized to implementing any particular type of algorithm. It is a truly general-purpose self-reconfigurable hardware device. Its application to the RGA is a demonstration of its versatility and suitability to a wide range of parallel problems [Mac99].

Moreover, other general-purpose single chip solutions are still, architecturally, composed of two subsystems: a controller and a reconfigurable platform [Nag98]. On the PIG, identical hardware is used for both of these subsystems. In the RGA implementation, the devices which implement the evolving members are the same as the devices which implement the control circuits for executing the

RGA. This can not be achieved with, say, and FPGA, since the inputs and outputs within the gate array can not directly read or write the chip's configuration registers.

Furthermore, since the evolved circuits use the same hardware as that used to implement the RGA, these evolved circuits potentially have the ability to create and modify other circuits. This means the PIG can be used to study evolvable hardware where the target circuit is itself configuring other circuits within the PIG.

The uniformity of the PIG's hardware has numerous advantages, of which only a few have been exploited in the present work. Further advantages include fault tolerance, ease of manufacture, and adaptability to new technologies. Hopefully, the field of evolvable hardware will find more applications for the PIG, leading to further research on the PIG itself, and ultimately the realization of a large-scale PIG.

Acknowledgment

The author wishes to thank Lisa Durbeck for many valuable discussions, ideas and suggestions during the course of this work, as well as for her continuous support and contributions to the PIG.

References

[Act95] Actel, *FPGA Data Book and Design Guide*, Actel, Sunnyvale, CA, 1995.

[Dre86] E. Drexler, *Engines of Creation*, Anchor Press/Doubleday, Garden City, NY, 1986.

[Hem94] H. Hemmi, J. Mizoguchi and K. Shimohara, "Development and Evolution of Hardware Behaviors," in R. Brooks and P. Maes, editors, *Proceedings of Artificial Life IV*, pages 371-376, MIT Press, 1994.

[Hig94] T. Higuchi, H. Iba and B. Manderick, "Evolvable Hardware," in H. Kitano and J. Hendler, editors, *Massively Parallel Artificial Intelligence*, pages 398-421, The AAAI Press, Menlo Park, CA, 1994.

[Kaj98] I. Kajitani et al., "A Gate-Level EHW Chip: Implementing GA Operations and Reconfigurable Hardware on a Single LSI," in M. Sipper, D. Mange and A. Pérez-Uribe, editors, *Evolvable Systems: From Biology to Hardware*, pages 1-12, Springer, 1998.

[Kit96] H. Kitano, "Morphogenesis for Evolvable Systems," in E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware: The Evolutionary Engineering Approach*, volume 1062 of *Lecture Notes in Computer Science*, pages 99-117, Springer, 1996.

[Koz92] J. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, 1992.

[Koz98] J. Koza et al., "Evolving Computer programs using Rapidly Reconfigurable Field-Programmable Gate Arrays and Genetic Programming," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 209-219, ACM Press, New York, 1998.

[Len93] C. Lent, P. Tougaw and W. Porod, "Bistable Saturation in Coupled Quantum Dots for Quantum Cellular Automata," *Applied Physics Letters 62*, pg. 714, Feb 1993.

[Lip96] R. Lipton and E. Baum, editors, *DNA Based Computers*, American Mathematical Society, 1996.

[Mac99] N. Macias., "The PIG Paradigm: The Design and Use of a Massively Parallel Fine Grained Self-Reconfigurable Infinitely Scalable Architecture," *Proceedings of The First NASA/DOD Workshop on Evolvable Hardware (EH'99)*, 1999.

[Nag98] K. Nagami, K. Oguri, T. Shiozawa, H. Ito and R. Konish, "Plastic-Cell Architecture," in K. Pocek and J. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, IEEE Society, Los Alamitos, CA, 1998.

[Sip97] M. Sipper., *Evolution of Parallel Cellular Machines*, volume 1194 of *Lecture Notes in Computer Science*, pages 119-127, Springer, 1997.

[Tom96] M. Tomassini, "Evolutionary Algorithms," in E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware: The Evolutionary Engineering Approach*, volume 1062 of *Lecture Notes in Computer Science*, pages 19-47, Springer, 1996.