An adaptive FPGA and its distributed routing

Yann Thoma and Eduardo Sanchez
Ecole Polytechnique Fédérale de Lausanne (EPFL)
Logic Systems Laboratory
CH-1015 Lausanne, Switzerland
Email: name.surname@epfl.ch

Abstract— **Every commercially available FPGA supplies high routing capabilities. However, placement and routing are processed by a computer before being sent to the chip. This non-adaptive feature does not fit well with bio-inspired applications such as growing systems or neural networks with changing topology. Therefore we propose a new kind of routing, built in hardware and totally distributed. Unlike previous works about routing, our approach does not need a central control over the process. In this paper we present a new FPGA embedding this algorithm, as well as the basic idea of our architecture, based on a parallel implementation of Lee shortest path algorithm. We then present a second algorithm that decreases the number of possible congestions, a third that reduces the execution time, and a fourth that combines both techniques. Finally we introduce different neighborhoods and compare all these algorithms in terms of area, speed, path length and congestion.**

I. Introduction

Since the appearance of the first FPGA [5], the routing complexity of such devices has continuously increased. These routing capabilities are however static in the sense that a computer has to create a routing scheme in a placement and routing (P&R) process before configuring the FPGA. During this operation, a software tries to minimize the delays between the modules put into the logic elements of the device. This approach has the advantage of providing a general view over the entire configuration, with the possibility of optimizing both the area and the speed of the design.

After configuration, the FPGA possesses a functionality, characterized by the content of the logic elements (typically look-up tables, registers and/or logic gates) as well as by their connectivity. Some applications such as JBits [8] for Xilinx devices allow a user to partially reconfigure a part of the FPGA, but with the risk of burning the device in case of short circuits. Moreover this reconfiguration is applied serially by a computer connected to a JTAG port, and is therefore slow and centralized, not managed by the FPGA itself.

This lack of adaptability, in terms of functionality and routing, does not fit well with bio-inspired applications. Life is characterized by its inherent plasticity, at different levels: species evolve to better adapt to their environment, individuals learn how to respond to specific stimuli, and self-healing capabilities allow an organism to survive after an attack. As living beings are composed of cells, bio-inspired systems are often built on cellular systems, and to allow plasticity, the cell functionality and the intercellular routing should be adaptable.

Cellular systems implemented on FPGAs could possibly need to create data paths at runtime. For instance, a neural network could change its connectivity to adapt to new conditions or to self-organize [2], or a cellular artificial organism could grow from a single cell up to an entire organism [16], a first step to self-repair. Therefore, such applications need a distributed routing system managed by the FPGA itself, without external intervention, letting cells initiating new routings. The basic blocks of such system should also be modifiable, for functionality adaptation and self-repair. Furthermore, this system would have to be electrically safe, avoiding short circuits that could damage the chip.

To reduce this lack of adaptability, a chip called POEtic [13] has been developed, based on an array of logic elements with a basic connectivity, and a second layer composed of an array of routing units, implementing a distributed routing algorithm [18]. In this paper we briefly present this algorithm, HIDRA, for Hardware Incremental Distributed Routing Algorithm, and propose some improvements to avoid congestion problems that can appear when the density of connections is too high. A third algorithm, whose aim is to reduce the execution time of a routing process is also presented, as well as a fourth one being a mix of both new approaches. Unlike previous work about parallel routing, these four algorithms have been implemented with a neighborhood of 3, 4, 6 and 8, in order to find the best candidates under constraints of connection density, number of pins and area available. The new algorithms are then compared in terms of speed, area required and congestion, with the one of POEtic.

The next section briefly presents the main innovative features of the reprogrammable part of the POEtic chip. Then, in section III we present some hardware implementations of the routing problem. Section IV shows our hardware implementation, and all the basics of our approach. Then, chapter V proposes a new algorithm, that can fit to software or hardware, and whose purpose is to minimize the congestion problem. Section VI shows a line-search approach in order to reduce the execution time, as well as an attempt to reduce the execution time and the congestion. Section VII presents different neighborhoods that can be used. We then show the experiment that has been used to compare algorithms and neighborhoods in terms of speed, congestion, path length and area. We finally conclude by a summary of the advantages of each approach.

II. POETIC CHIP REPROGRAMMABLE PART

The goal of the POEtic chip is to ease the implementation of bio-inspired applications involving evolution (Phylogeny), development (Ontogeny), and/or learning (Epigenesis). It is mainly composed of a full custom microprocessor and a reconfigurable part. The first one is a 32-bit RISC with a parallel access to the second part. All details about the chip can be found in [13], and we will just focus here on the specific features of POEtic, before going further into the distributed routing.

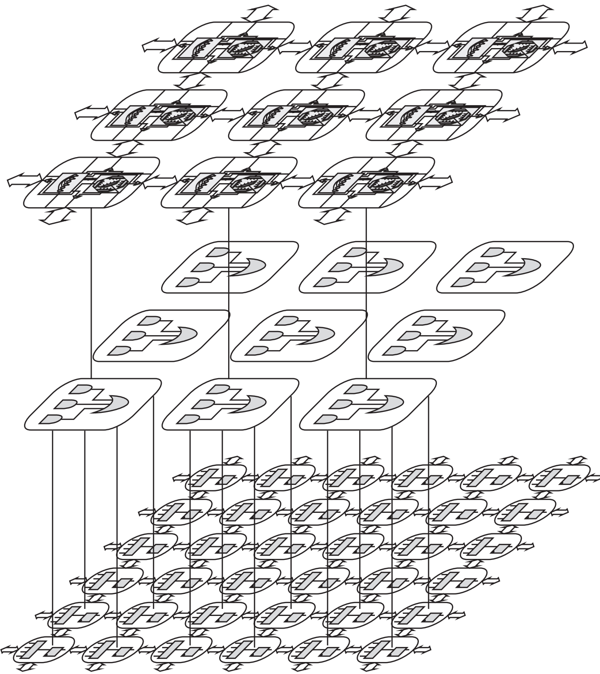

Fig. 1. Our architecture, based on a layer of programmable logic elements at the bottom, an interface, and a top layer of routing units.

The reconfigurable part of the chip (Figure 1) is composed of a plane of basic programmable elements called molecules, and a plane of routing units that implement our distributed routing algorithm. A molecule contains basically a 4-LUT and a flip-flop, the output being combinational or sequential (Figure 2). The intermolecular communication is realized by the way of switch boxes. Each switch box consists of eight input lines (two from each cardinal direction) and eight corresponding output lines, and are implemented with 8-input multiplexers. Two outputs are sent into each of the four neighbours of the molecule, as shown in figure 2.

A molecule can be used in eight different operational modes, and can be, for instance, a 4-LUT, two 3-LUT, a 16-bit shift register, an output or an input to the routing plane, or can serve to configure other molecules. This last mode is one of the new features of POEtic, and allow a molecule to partially reconfigure other molecules of the chip, with a serial access to the configuration bits. This can be very useful for evolutionary or self-repair mechanisms, where the cells of an organism can dynamically change their behavior by modifying the content

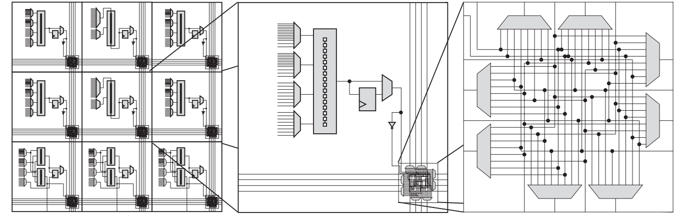

Fig. 2. Structure of a molecule.

of LUTs, or all configuration bits of some molecules. We can also notice that no configuration can create short circuits, permitting to use POEtic for evolvable hardware applications.

Finally, the dynamic creation of data paths is the last specific feature of POEtic. The molecules can launch routing processes to configure the multiplexers of the routing units, allowing cells to connect at runtime.

III. BACKGROUND ON HARDWARE ROUTING

The goal of a routing algorithm is to find a path between a starting and a final point, or to find a way to connect different points in a graph. It can be to link one source to many targets (broadcast), or many sources to many targets (in an FPGA for instance), but it often has to minimize the length of paths created (in electronic circuits, we want to reduce the delay propagation, just like we want to minimize the time of a trip by car between two cities). Therefore, the shortest path problem is closely related to routing algorithms.

In 1959, Dijkstra proposed an algorithm to solve the shortest path problem [7] in a graph where nodes are connected with edges of positive weights. Giving two points in the graph, the algorithm finds the shortest path between them in a time $O(n^2)$. Two years later, Lee designed an algorithm [11] to find the shortest path in a two-dimensional array of units, each one of them being connected to its 4 neighbors. This is basically a simplified version of Dijkstra's in the sense that a 2D array can be considered as a regular graph with weights equal to 1. The idea is to define a source and a target, and let an expansion phase start from the source. Every point waits for the expansion wave, and when it is reached, it stores its distance to the source. When the target is found, a traceback process follows the distances from n (distance of the target) to 0 (the source). Figure 3 illustrates the result of this process for a target with distance 3. We can notice that this algorithm is sequential, as only one node is being expanded at a time.

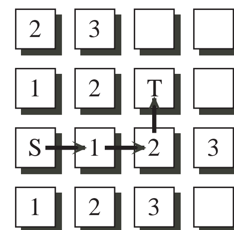

Fig. 3. In Lee algorithm, after having reached the target, the path between the source and the target is created by the traceback phase.

Later, in 1968, Mikami and Tabuchi introduced another algorithm [12], based on a line-search system. The expansion phase differs from Lee's, the source expanding lines instead of just reaching the first four neighbors. Therefore, in one step, every point on the same horizontal or vertical line than the source is reached. Subsequently every reached point expands lines as well, until the target is found. This algorithm always finds a path if it exists, but not necessarily the shortest. Its real advantage arises with a parallel implementation, where a line is expanded in one clock cycle, as we will explain in section VI. Other approaches have also been proposed, but are out of the scope of this article.

Based on the same idea of Lee's algorithm, parallel implementations have been proposed. In 1996, Adamatzky [1] and Hochberger [9], separately solved the shortest path problem with cellular automata.

Some hardware implementation of Lee algorithm have been realized, based on SIMD, MIMD, analog devices, or digital devices, to exploit the inherent parallelism of this algorithm. An FPGA with hardware-assisted routing has been created by DeHon et al. [6], but the interconnection network is too simple for really adaptive applications. Before that, four digital chips have been proposed, by Breuer [3], Iosupovicz [10], Ryan [17], and finally by Nestor [15]. All of these approaches are based on cells, each one representing a cell of figure 3. While these systems deal with cells connected by non-directional links and are useful for the routing of PCB wires, in 2001, Moreno introduced a hardware design of an algorithm for routing directional links [14]. It is based on routing units connected to 4 neighbors, and implementing a routing algorithm, with a small amount of logic required. Here again, as for every other implementation, the algorithm starts with a source and target defined by an external agent. However, the new idea is to configure multiplexers that could be used later to connect different parts of an electronic device. This approach was intended to create a routing algorithm for bio-inspired hardware like an electronic substrate on top of which applications such as adaptive neural networks could be implemented.

All of these parallel implementations follow the same principle of having cells for the expansion and traceback phases, and an external agent (a controller or configuration bits) to control which cells have to be connected. In cellular systems however, a good point would be to let the cells connect themselves without any external control, to let a system grow, or self-repair. We therefore introduce a new totally distributed algorithm, where the cells are responsible for connecting to each other. We base our approach on identifiers (ID), a source having an ID, and a target knowing the ID of its corresponding source.

IV. Hardware Implementation: HIDRA

This algorithm was originally designed for the POEtic chip. The molecules in mode *input* or *output* store an ID in the LUT, used as a 16-bit shift register. A routing unit is connected to four molecules via an interface and access the ID in a serial manner. This interface, while connecting 4 logic elements in our physical implementation (Figure 1), could easily be changed to deal with 9 elements or more, depending on the density of routing units that has to be achieved.

We now describe the distributed routing algorithm HIDRA, as it is realized in POEtic, before showing some possible improvements. The basic algorithm was designed for a 4-neighbors system, because of constraints of size and number of pins. We will show later that an 8-neighborhood is more efficient, and would be the best solution for a bigger chip.

A. Routing Units

First of all, let's fix the global architecture of a routing unit, as only its controller will change for the different algorithms. We assume that the identifier of a source or a target is stored outside the routing unit, in order not to depend on the identifier size. This identifier will be accessed in a serial manner, by sending a read enable. A second external signal, a trigger, should be supplied to the routing unit. It can be the same for all units, and is managed by the controller of any of these units, and serves to indicate the number of clock cycles needed to compare two identifiers. For instance, if we deal with 16-bit identifiers, then when enabled, the trigger should supply a '1' every 16 clock cycles. Finally, since the goal is to link logic elements dynamically, a signal can be sent from the logic element to the routing unit, and vice versa, and a signal from the logic element has to indicate if the routing unit has to act as nothing special or an input or output. In this case, another signal selects if it has to initiate a connection or if it just waits for another one to launch the routing process (an output of a cell won't necessarily need to be connected, while an input has to retrieve data from somewhere). Finally, a feedback signal indicates to the logic element if it is connected through the routing layer or not.

Figure 4 shows the rough structure of a routing unit, as well as its major outputs. We can observe that two wires send data to each direction. The one called `val_out` is used during the routing process to send signals from the controller to the neighbors, while under normal operation it simply sends the value selected from the switchbox. Four multiplexers are used for this purpose and managed by the controller (in the initial state, the switchbox output is selected, while in all others states, the value from the controller has the priority). The one called `prop_out` serves to propagate values through the entire array. It implements a priority mechanism that selects where to propagate the signal at '1' depending on its origin. If a '1' comes from the south only the north output will be '1', conversely if a '1' comes from the west, the north, south and east will be '1', and so on. The interesting feature of this system is that it creates a priority of the most bottom-left propagation unit. Consequently, if many routing units propagate a '1', the most bottom-left will be aware of its special position, allowing the selection of one unit out of many.

The switchbox is composed of five multiplexers (MUX), one pointed to each direction, and one to select a value to be sent to the element connected to the routing unit. Each MUX

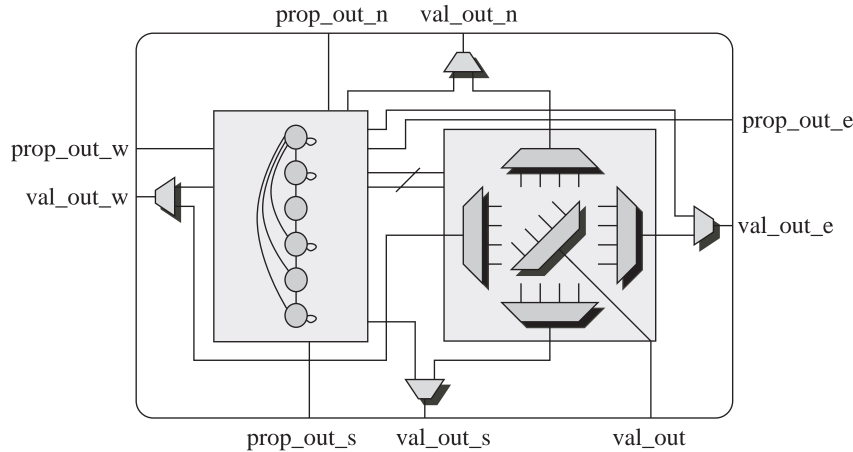

Fig. 4. The general architecture of a routing unit, showing the controller and the switchbox.

connected to a neighbor has four inputs, i.e. one from each other direction, and one from the logic element connected to the routing unit, while the MUX to the logic element can select from any of the four neighbors input. Every multiplexer is controlled by three configuration bits managed by the state machine. Two of them really define the behavior of the multiplexer, while the third indicates if the multiplexer is configured or not, that is, if it is part of a path or not. The controller has to deal with these three bits during the expansion phase, in order not to erase an existing path. Moreover it is used to follow an existing path from the source by reusing it.

B. Controller

All components described previously are similar for our different algorithms, the controller being the only change. This controller is built with a finite state machine of 6 states, a propagation unit, and a serial comparator to compare identifiers. Our basic distributed algorithm is based on different phases that we will explain in this section.

- **Phase 1.** When in initial state, a routing unit wanting to be connected sends a '1' through the propagation line. If more than one routing unit wants to start a routing process, the propagation unit allows the controller to know whether it is the most bottom-left or not. This operation is executed in one clock cycle, the propagation being combinational.
- **Phase 2.** After this step, the most bottom-left is considered to be the master for this routing process, and starts sending its identifier in serial through the propagation line. Every routing unit, being a source or a target, sends an enable to its external element to shift the identifier and the trigger. Every source or target compares the received identifier bit with its own identifier, until the trigger sends a '1'. After these n clock cycles the sources and targets know whether they have to be involved in the current routing process or not, that is if they have the same identifier as the master.
- **Phase 3.** During the next step, lasting one clock cycle, the master sends a '1' through the propagation line if it is a source. In this case, all sources that detected the same identifier are disabled, in order to make sure that the master source, and only that one, will be the one connected. If the master is a target, then all other targets with the same identifier will be disabled. Actually, if targets did not disable, the routing of the entire chip could be faster. However we chose our option because in a cellular system, even if many cells have inputs with the same ID, only one may want to connect to another one.
- **Phase 4.** While the previous steps are identical for all algorithms presented in this paper, the next phase will differ for each particular algorithm. The expansion phase, in the basic implementation, is a parallel implementation of Lee's algorithm. The source involved in the process starts by sending a '1' to its four neighbors through the `val_out` port. At each clock cycle, a routing unit checks whether a signal arrives from a neighbor. In case of many inputs being active at the same time, priority is given to the north, then east, south and west. The origin of the signal is stored in two registers, and during the next clock cycle the routing unit sends a '1' to its neighbors. The controller checks, before sending the signal, if the multiplexer corresponding to the direction is already configured or not, in order not to destroy any existing paths. An exception occurs when the multiplexer is configured and selects the value coming from the origin of the expansion. In this case it means that it is part of a path starting from the current source, and it is a good point to reuse the existing path. If no existing path blocks up the new one to be created, then this phase only requires d clock cycles, d being the Manhattan distance between the source and the target.
- **Phase 5.** When the target is reached by the expansion phase, after one clock cycle, it propagates a '1' through the propagation line, indicating the end of the process, and sends a '1' to the neighbor being its origin. The

neighbor then propagates it to its own origin, and so on, until the source is reached. During this traceback phase, every routing unit on the path configures the corresponding multiplexer accordingly to its origin.

Once the routing process finished, a new one can start, until every routing unit wanting to be connected is connected. Then, the circuit can execute any task, using the paths created to send values through the chip. We can observe that at any time a routing unit can start a new routing process.

The physical implementation of the routing unit controller consists of 5 flip-flop and a 6-state finite state machine. The first register indicates if it is already connected as an input or an output, the second whether it is the master of the routing process, a third if it participates or not to the current process, and the last two indicate the origin of the expansion.

C. Congestion Problem

In some cases, where the density of sources and targets is too high, a congestion problem can occur, the expansion phase not being able to reach the target. In this case, a flag is set to '1' by one of the routing units, indicating that no solution is found. If we deal with the hypothesis of a microprocessor capable of accessing the reconfigurable array, then the microprocessor should try to change the place of some components to check if this could avoid this trouble.

V. Reducing Congestion: HIDRA-RC

As we will see later, with a 4-neighborhood, congestions appear quite fast, so there should be a way to reduce it. As the creation of paths corresponds to the configuration of multiplexers, a basic idea to reduce the congestions is to minimize the number of multiplexers used to connect sources and targets instead of only trying to minimize the paths length. This concept can be applied whenever more than one target is connected to a source, which is the case in many applications (in neural networks, the output of a neuron is often used by several neurons).

In 1986, Watanabe and Sugiyama [19] proposed a parallel routing algorithm very efficient in this sense, when many targets have to connect to the same source. It finds the quasi-minimum Steiner-Tree between all the points that have to be connected together. This algorithm is very interesting, but needs to know exactly all routing units that want to be connected, while in our approach we want to allow the creation of paths a posteriori, to have a fully dynamic system.

We therefore propose an adaptation of our basic algorithm capable of reducing the number of multiplexers involved in paths. While in the standard implementation the source starts the expansion phase, in this new algorithm, the source and every routing unit that is part of an existing path connected to this source launch the expansion. This way, the routing process finds the shortest path between the target and the existing path connected to the source. Figure 5 shows the difference between the standard algorithm HIDRA and the one reducing the congestion, HIDRA-RC (for HIDRA with Reduced Congestion). The numbers in the squares represent the time when the routing unit was reached by the expansion. In this situation we can observe that two multiplexers are saved for further routing process, because of the reuse of an existing path.

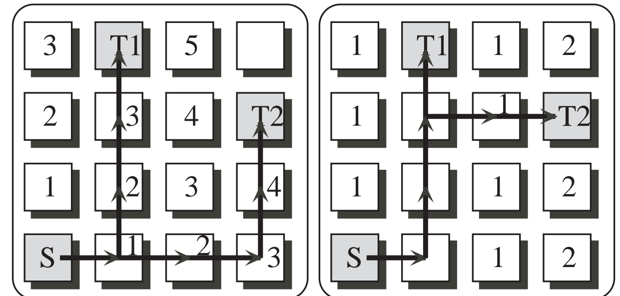

Fig. 5. On the left, the result with HIDRA. On the right, the improved version, HIDRA-RC.

With HIDRA-RC, we do not guarantee to find the shortest path considering the existing ones, but by reducing the number of multiplexers configured, we let more of them available for further routing process. We will see with the experiments that the improvement can be very important, making HIDRA-RC an excellent candidate for new chips implementations where the targeted applications have a high density of connections.

VI. Reducing Execution Time: HIDRA-RT and HIDRA-RTC

In HIDRA, the expansion phase needs at least $|x_t - x_s| + |y_t - y_s|$ clock cycles[1]. Following the idea of Mikami, and adapting its principle to a parallel implementation, this time can be dramatically reduced, by using lines extensions instead of simply having a front wave increasing every clock cycle.

The algorithm HIDRA-RT (for Reduced Time), only needed small changes in the state machine of the routing unit controller. Waiting for the expansion wave, if a routing unit is reached by a neighbors, then it propagates the signal to the opposite direction in a combinational manner. Then, the next clock cycle, it will propagate to the other directions, like in HIDRA. Figure 6 shows the first two steps of expansion, the arrows being the sense of propagation.

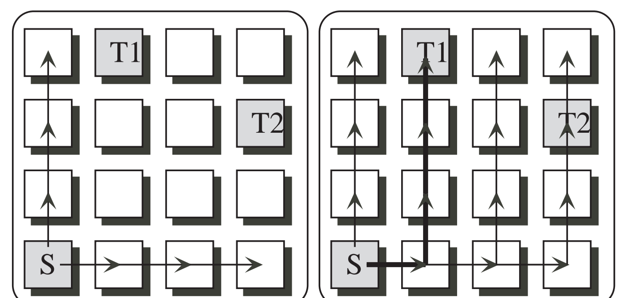

Fig. 6. In two steps of expansion, target 1 is reached.

[1]This corresponds to the Manhattan distance between the two points.

In an environment with a small number of existing paths, a new path would only need two clock cycles of expansion, because with an horizontal line followed by a vertical one, all points can be reached. It is also important to notice that this algorithm will always find a path between two points if such a path exists. However, it won't necessarily be the shortest one; which is the price to pay for a reduced execution time.

An extension to HIDRA-RT, HIDRA-RTC, has been also realized, as a mixture of HIDRA-RC and HIDRA-RT. It is based on line-search, but with the features allowing to reduce the potential number of multiplexers requisitioned. In this algorithm, the source and all routing units that are part of an existing path connected to this source launch the expansion, and then this expansion is made by sending lines, like in HIDRA-RT. A routing unit reached by the wave directly propagates it to the directions where multiplexers are configured to select the corresponding origin, and to the opposite direction of the origin. We will observe that this solution have congestion statistics between HIDRA and HIDRA-RT.

VII. Different neighborhoods

The physical implementation of HIDRA was realized with a 4-neighborhood, because of area and pins restrictions. However, we implemented the four previous algorithms with different neighborhoods. We chose the three regular tessellations, made of units of triangles, squares and hexagons, and the Moore neighborhood (8 neighbors) (Figure 7).

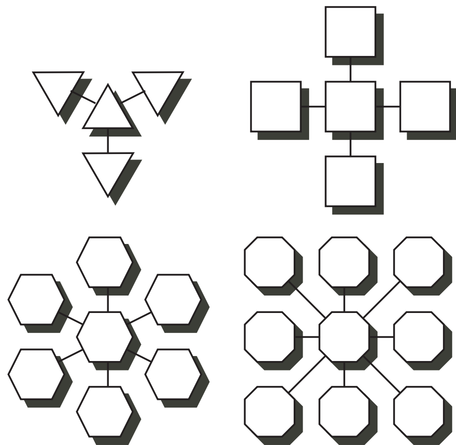

Fig. 7. The four neighborhoods implemented.

While previous works on parallel routing was made with a 4-neighborhood, we will show here that different ones can be more efficient. For instance, an 8-neighborhood with a density being a quarter of the one of a 4-neighborhood has the same efficiency in terms of congestion, while it uses half of the area in terms of silicon.

As far as the implementation of these neighborhoods is concerned, the state machine stays the same, only more logic being used to deal with more neighbors. The origin register, only requiring two bits for 3 and 4 neighbors also requires three bits for 6 and 8 neighbors. Finally the switchbox' size really makes the difference, as the number of multiplexers equals $n+1$ for a n-neighborhood, and as its size and number of configuration bits also depends on it. For instance, for an 8-neighborhood, the switchbox contains 36 configuration bits, while only 12 are required for a 3-neighborhood.

While the 8-neighborhood will show to be the best solution, it is important to bear in mind the number of pins that should be used to connect different chips together. Table I illustrates the number of pins for an array of X by Y routing units.

TABLE I

Number of pins for an array of X by Y routing units

Neighborhood	Pins
3	$2X + 4Y + 8$
4	$4X + 4Y + 8$
6	$8X + 8Y + 8$
8	$12X + 12Y$

These equations only take care of the signals `val_out`/`val_in`, `prop_out`/`prop_in`. An interesting feature comes from the fact than the `prop` signal can be shrunken into only one wire to pass from a chip to another, thus dramatically reducing the number of pins.

The number of pins required by the 8-neighbors solution could be an handicap if we would like to create a chip with an important number of routing units. However, in this case, not every routing unit on the border has to be connected to an external pin. This would reduce the total routing capacity of a multi-chip design, but this way we can put as many routing units as required inside a device.

VIII. Algorithms comparison

We presented four algorithms and four different neighborhoods, making sixteen ways of implementing distributed dynamic routing. As VHDL implementations have been developed for all of them, it is easy to compare them in terms of space, execution time, and congestion. Actually, for experiments requiring a huge number of runs, we used a software implementation of each algorithm instead of Modelsim simulations in order to reduce the simulation time. This shortcut has been checked to make sure it acts exactly like the parallel implementation, and that the evaluations of clock cycles elapsed are identical.

The results of these experiments offers a lot of data to analyze, to compare algorithms and neighborhoods. First of all, we have a look at the number of transistors of each implementation. Then the three following subsections show the most interesting result highlighting the main differences concerning the execution time, the path lengths, and the congestion.

A. Transistors

As the goal of our approach is to create hardware implementations of routing algorithms, the number of transistors needed for each one is a very important factor, helping to choose one solution among the others. A VHDL description of all of the 16 solutions has been developed, at the RTL

level, and synthesized with Leonardo and a special set of basic elements to fit in, counting the number of needed transistors. We assume 2 transistors for an inverter, 4 for a NAND gate, and 20 for a register. We chose to make a comparison in terms of transistors instead of CLBs or LEs of commercial FPGAs, as the aim of our algorithms is to be physically implemented more than being put onto an FPGA. Table II illustrates this number for every algorithm and neighborhood.

TABLE II
NUMBER OF TRANSISTORS OF A ROUTING UNIT FOR EACH IMPLEMENTATION (TRANSISTORS/REGISTERS)

	3	4	6	8
HIDRA	736/23	884/26	1476/40	1952/48
HIDRA-RC	862/23	1078/26	1826/40	2422/48
HIDRA-RT	772/23	958/26	1592/40	2070/48
HIDRA-RTC	834/23	1048/26	1748/40	2240/48

Firstly, we can notice than the smallest solution is the one that has been physically implemented, as the area was the main issue in this project. Secondly, we can see that the 8-neighbors solution uses approximately twice the area of the 4-neighbors one, which will be very important in the conclusion. Thirdly, we can observe that the switchbox occupies a large amount of resource (from 36.5% to 58.4%), letting the rest for the controller.

B. Execution Time

The evaluation of execution time is based on the number of clock cycles needed to complete the routing process. In order not to introduce a bias due to congestion, we divide the total number of clock cycles by the number of routed paths. This way we obtain the average number of clock cycles for the creation of one path. We can notice that the difference between the algorithms is not really significant, due to the fact that only the expansion phase differs from one to another. As we showed before, there is a kind of latency for every routing process, because of one clock cycle is needed for defining a master, 16 for the identifier (in our case), one to know if a source or a target launched the process, and at the end, one to create the path. So there are at least 19 clock cycles, the rest being the expansion phase.

Table III shows the average number of clock cycles for each algorithm with a 4-neighborhood, on a array of size 20×20, with or without the latency period.

TABLE III
EXECUTION TIME WITH A 4-NEIGHBORHOOD

Algorithm	With latency	Without latency
HIDRA	34.36	14.98
HIDRA-RC	30.41	11.41
HIDRA-RT	23.01	4.01
HIDRA-RTC	22.66	3.66

C. Path Length

With regard to the length of created paths, the definition of our algorithms allows to predict that HIDRA performs better than the others, since the line search approach does not necessarily find the shortest path and the reduction of congestion implies a potential lengthening of the paths. However, it is very important to notice that HIDRA-RC creates paths with less than one more multiplexer than HIDRA. This difference can be considered as relatively insignificant compared to the advantage in terms of congestion. Table IV shows the average length of a path in our experiments with three targets per source, for an array of 20×20.

TABLE IV
AVERAGE PATH LENGTH, DEPENDING ON THE ALGORITHM AND THE NEIGHBORHOOD

	3	4	6	8
HIDRA	20.68	14.98	12.66	9.86
HIDRA-RC	20.85	15.35	13.39	10.70
HIDRA-RT	28.91	15.42	15.03	12.85
HIDRA-RTC	28.92	15.67	15.39	13.50

One can observe that the higher the neighborhood, the shorter the paths. This is due to the fact that in an 8-neighborhood for instance, diagonals allow to reduce the length to $max(|X_t - X_s|, |Y_t - Y_s|)$ if there is no existing path on the way. However, this advantage requires the use of bigger multiplexers in the case of 6 and 8 neighbors. Therefore the propagation time of a signal won't be significantly better with 6 and 8 neighbors.

D. Congestion

With regard to congestion, the bigger the neighborhood, the more we avoid this problem, because there are more possibilities to find a way from a point to another. Figure 8 shows the number of congestions within 200 runs, with the HIDRA algorithm, for the four different neighborhoods and three targets per source.

The interesting fact of this graph is that we can observe that for neighborhoods 3, 4, 6 and 8 the performance starts degrading respectively at points 25, 50, 100 and 200. For a total number of 20x20=400 routing units, we can deduce that for a random placement of sources and targets, the acceptable density of targets is 1/16, 1/8, 1/4 and 1/2, if three targets are connected to one source. It is more interesting to observe that even with different numbers of targets per source the factor 2 between the neighborhoods is preserved, showing that the 8-neighborhood seems to be a very good candidate.

Let's now have a look at the different algorithms within the same neighborhood. We compare the four algorithms with the 8-neighborhood, this one being the best in terms of congestion. As there would be no difference between HIDRA and HIDRA-RC with one target per source, we choose to deal with 3 targets per source [2]. Figure 9 shows the number of congestions for a

[2]The same results appear with 2 or 4 targets per source.

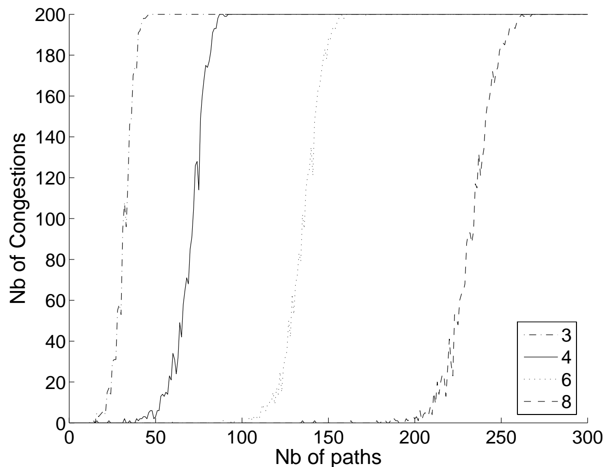

Fig. 8. Number of congestions for each neighborhood, depending on the number of paths to create, with HIDRA.

number of targets ranging from 100 to 300.

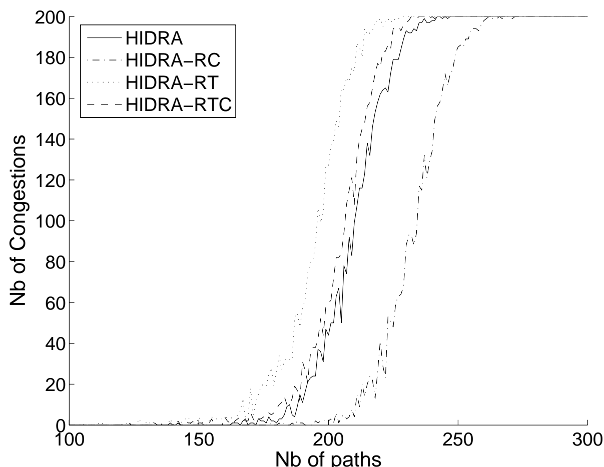

Fig. 9. Number of congestions for each algorithm, with 8 neighbors, depending on the number of paths to create.

As expected, HIDRA is more efficient than HIDRA-RT, as it always finds the minimal existing path. There is also a huge difference between HIDRA and HIDRA-RC, the second avoiding the congestion problem for many more runs. Finally, it is interesting to observe that HIDRA-RTC shows almost the same behavior as HIDRA. As HIDRA-RTC is faster, if the small amount of additional logic required, it can be a good alternative to HIDRA.

IX. Conclusion

In this paper we described the HIDRA algorithm and all its derivatives to reduce congestion or execution time, as well as different neighborhoods. Thanks to the results obtained with our experiments, we are now capable of choosing an option depending on the number of transistors and pins available and on the projected density of connections.

The major result consists of the comparison between 4 and 8 neighbors. Keeping in mind that there is a factor 2 in terms of number of transistors between 4 and 8 neighbors, and that there is a factor $1/4$ in terms of congestion, we can argue that it is better to have a design with $n \times n$ number of routing units of 8 neighbors instead of having $2n \times 2n$ routing units of 4 neighbors. In this configuration, the number of transistors with 8 neighbors is divided by 2, and the congestion troubles are dramatically reduced by a factor 2.

We can also consider HIDRA-RTC as an excellent candidate for applications that need a fast routing. This algorithm has almost the same congestion characteristics than HIDRA, but can really reduce the execution time, with only a small amount of additional logic.

With regard to congestion, HIDRA-RC appeared to be the best solution, allowing a better density of sources and targets. Therefore, for applications with an important number of connections required, the best approach would be HIDRA-RC with an 8-neighborhood.

Finally we can observe that the congestion curves show a phase transition between ”everything can be routed” to ”nothing can be routed”. Such transition is exactly what can be seen in percolation models [4], and we are currently working on a characterization of our algorithms from this point of view.

X. Acknowledgements

This project was funded by the Future and Emerging Technologies programme (IST-FET) of the European Community, under grant IST-2000-28027 (POETIC). The information provided is the sole responsibility of the authors and does not reflect the Community’s opinion. The Community is not responsible for any use that might be made of data appearing in this publication. The Swiss participants to this project are supported under grant 00.0529-1 by the Swiss government.

References

[1] A. I. Adamatzky. Computation of shortest path in cellular automata. *Mathematical and Computer Modelling*, 23(4):3415–3418, 1996.

[2] S. Bornholdt and T. Rohlf. Topological evolution of dynamical networks: Global criticality from local dynamics. *Physical Review Letters*, 84(26):6114–6117, June 2000.

[3] M. A. Breuer and K. Shamsa. A hardware router. *Journal of Digital Systems*, 4(4):393–408, 1981.

[4] S. R. Broadbent and J. M. Hammersley. Percolation processes I. crystals and mazes. *Proceedings of the Cambridge Philosophical Society. Mathematical and Physical Science*, 57:629–641, 1957.

[5] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field Programmable Gate Arrays*. Kluwer Academic Publishers, 1992.

[6] A. DeHon, R. Huang, and J. Wawrzynek. Hardware-assisted fast routing. In *Proceedings of the 10 th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM’02)*, page 205. IEEE Computer Society, 2002.

[7] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[8] S. A. Guccione, D. Levi, and P. Sundararajan. Jbits: A java-based interface for reconfigurable computing. In *2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 1999.

[9] C. Hochberger and R. Hoffmann. Solving routing problems with cellular automata. In S. Bandini and G. Mauri, editors, *Proc. 2nd Conference on Cellular Automata for Research and Industry, ACRI ’96*, pages 89–98. Springer, 1996.

[10] A. Iosupovicz. Design of an iterative array maze router. In *Procs. IEEE International Conference on Circuits and Computers (ICCC)*, pages 908–911. IEEE, October 1980.

[11] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, September 1961.

[12] K. Mikami and K. Tabuchi. A computer program for optimal routing of printed circuit conductors. In *IFIP Congress*, volume 2, pages 1475–1478, 1968.

[13] J.-M. Moreno, Y. Thoma, E. Sanchez, O. Torres, and G. Tempesti. Hardware realization of a bio-inspired POEtic tissue. In R. S. Zebulum, D. Galtney, G. Hornby, D. Keymeulen, J. Lohn, and A. Stoica, editors, *Proc. 2004 NASA/DoD Conference on Evolvable Hardware*, pages 237–244, Los Alamitos, California, 2004. IEEE Computer Society.

[14] J. M. Moreno Arostegui, E. Sanchez, and J. Cabestany. An in-system routing strategy for evolvable hardware programmable platforms. In *Proc. 3rd NASA/DoD Workshop on Evolvable Hardware*, pages 157–166. IEEE Computer Society Press, 2001.

[15] J. A. Nestor. A new look at hardware maze routing. In ACM, editor, *Proc. 12th ACM Greate Lakes Symposium on VLSI (GLSVLSI '02)*, pages 142–147, New York, USA, April 2002.

[16] D. Roggen, Y. Thoma, and E. Sanchez. An evolving and developing cellular electronic circuit. In J. Pollack, M. Bedau, P. Husbands, T. Ikegami, and R. A. Watson, editors, *Proc. Ninth International Conference on the Simulation and Synthesis of Living Systems (ALIFE9)*, pages 33–38, Cambridge, Massachusetts, USA, 2004. The MIT Press.

[17] T. Ryan and E. Rogers. An ISMA Lee router accelerator. *IEEE Design and Test of Computers*, 4(5):38–45, October 1987.

[18] Y. Thoma, E. Sanchez, J.-M. Moreno Arostegui, and G. Tempesti. A dynamic routing algorithm for a bio-inspired reconfigurable circuit. In P. Y. K. Cheung, G. A. Constantinides, and J. T. de Sousa, editors, *Proc. of the 13th International Conference on Field Programmable Logic and Applications (FPL'03)*, volume 2778 of *LNCS*, pages 681–690, Berlin, Heidelberg, 2003. Springer Verlag.

[19] T. Watanabe and Y. Sugiyama. A new routing algorithm and its hardware implementation. In *Proc. 23rd ACM/IEEE conference on Design automation*, pages 574–580, Piscataway, NJ, USA, 1986. IEEE Press.