Genetic Programming and Evolvable Machines (2019) 20:155–186
https://doi.org/10.1007/s10710-018-9340-5

On the scalability of evolvable hardware architectures: comparison of systolic array and Cartesian genetic programming

Javier Mora[1] · Rubén Salvador[2] · Eduardo de la Torre[1]

Received: 14 February 2018 / Revised: 13 August 2018 / Published online: 1 October 2018

Abstract
Evolvable hardware allows the generation of circuits that are adapted to specific problems by using an *evolutionary algorithm* (EA). *Dynamic partial reconfiguration* of FPGA LUTs allows making the *processing elements* (PEs) of these circuits small and compact, thus allowing large scale circuits to be implemented in a small FPGA area. This facilitates the use of these techniques in embedded systems with limited resources. The improvement on resource-efficient implementation techniques has allowed increasing the size of processing architectures from a few PEs to several hundreds. However, these large sizes pose new challenges for the EA and the architecture, which may not be able to take full advantage of the computing capabilities of its PEs. In this article, two different topologies—*systolic array* (SA) and *Cartesian genetic programming* (CGP)—are scaled from small to large sizes and analyzed, comparing their behavior and efficiency at different sizes. Additionally, improvements on SA connectivity are studied. Experimental results show that, in general, SA is considerably more resource-efficient than CGP, needing up to 60% fewer FPGA resources (LUTs) for a solution with similar performance, since the LUT usage per PE is 5 times smaller. Specifically, 10×10 SA has better performance than 5×10 CGP, but uses 50% fewer resources.

Keywords FPGA · Evolvable hardware · Dynamic partial reconfiguration · Systolic array · Cartesian genetic programming · Scalability

1 Introduction

Evolvable hardware (EHW) is a design paradigm that uses a configurable hardware substrate to implement a circuit able to solve a specific problem. Unlike classical circuit design, where the parameters and architecture of the circuit are specified by

✉ Javier Mora
javier.morad@upm.es

Extended author information available on the last page of the article

the designer based on knowledge of the problem to solve, EHW relies on an *evolutionary algorithm* (EA) to generate a solution. This EA adjusts the parameters or structure of EHW based on a set of *training samples*.

These training samples are representative examples of the problem to solve. For instance, a system whose purpose is to remove a certain type of noise from an image stream would typically use a noisy image as a training input and the same image without noise as a training reference; likewise, a system whose purpose is to perform edge detection on an image would use a normal image as training input and the result of applying a known edge detection algorithm (which can be done in software) for the training reference. The EA would automatically generate HW configurations featuring different circuits (or processing behaviors) so that the result of processing the training input with each of these candidate solutions is as similar as possible to the training reference. Once a suitable solution has been found for the problem at hand, the EHW system shall be able to process new input for which the reference is unknown. This way, EHW can be used to create *self-adaptive* systems.

This evolutionary process can be *extrinsic* or *intrinsic*, depending on whether the candidate solutions are evaluated on a simulated model or on the EHW system itself. An advantage of intrinsic evolution is that it enables the system to be *self-healing*, as it is able to recover from faults in its fabric by evolving in order to find alternative solutions where these faults have a smaller effect, making the hardware *fault tolerant*. Additionally, it simplifies the requirements of the whole EHW system, which can be embedded in an SoC, making it completely *autonomous*.

FPGAs are a very good platform for implementing EHW, specially those with *dynamic partial reconfiguration* (DPR) capabilities. DPR is a process through which the FPGA can autonomously reconfigure part of its logic while the rest continues operating. This can be used to implement an EHW system as a piecewise circuit structured as an array of multiple *processing elements* (PEs), each of which implements a simple function, which can be individually changed by replacing the PE with a different one using DPR. This change in functionality will be driven by the EA, which will perform random small changes on the EHW circuit, evaluating how these changes affect the EHW performance and deciding whether to keep or revert these changes.

As larger FPGAs with more available resources appear and new design techniques that reduce the size of these PEs are adopted, the number of PEs that an EHW system can have increases. As an example, the system described in [30] implemented 16 PEs; however, new implementation techniques have allowed reducing the PE size by a factor of 20 and using a larger amount of resources from the FPGA, leading to the implementation of a similar architecture with a size of up to 576 PEs, which will be described later on this article. This increase in size opens the door to new research in the field of the scalability of EHW.

In this article, two EHW topologies are studied: the **systolic array** (SA) and **Cartesian genetic programming** (CGP). The aim of this article is to analyze and compare both topologies in terms of scalability and resource usage, in order to determine which architecture is more advantageous in each situation. For this purpose, a specific use case—an EHW-based image filtering application—has been studied, using both topologies at different sizes and analyzing the performance of the resulting solutions. This use case has been chosen due to its extensive use in previous work [5, 21, 25, 31, 39]; nevertheless, the conclusions obtained for this use case are expected to be extensible to other applications.

The rest of the article is organized as follows: before introducing the state of the art, the two topologies analyzed in this paper—SA and CGP—are described in Sect. 2, and their advantages and disadvantages in terms of scalability and resource usage are briefly discussed. Section 3 presents the state of the art, where multiple previous approaches in the field of EHW are surveyed. In Sect. 4, an EHW system based on an SA is described and evaluated. Section 5 does the same with an identical system based on CGP rather than SA, comparing their performance. Then, a modification in the SA architecture is proposed in Sect. 6 in order to improve its performance for large sizes. Finally, Sect. 7 summarizes the advantages and disadvantages of each architecture and their behavior with scalability, and briefly describes some of the possible future lines of development and potential applications.

2 Two topologies for EHW: CGP and systolic array

As stated in Sect. 1, EHW is commonly based on multiple PEs interconnected in a specific way. Each of these PEs has a certain number of inputs (usually two) which can be connected to either the *primary inputs* (the main inputs to the EHW system) or to the output of other PEs, implements a certain operation on the data it receives, and sends the processed result to other PEs, typically registering this output in order to create a pipelined data processing architecture.

Allowing every PE to get its inputs from any other possible PE in the system would lead to excessively complex routing (which is generally undesirable in FPGA design) and to an increased complexity of the search space, which could make it far more difficult, or even impossible, for the EA to find a suitable solution. Additionally, it would require using multiplexers of a very large size or another kind of routing mechanism in order to allow each PE to get its input from the chosen source. Therefore, typical EHW implementations limit PE connectivity, allowing only a few possible interconnections.

2.1 Cartesian genetic programming

Probably, the most used topology in EHW is *Cartesian genetic programming* (CGP) [22, 23], which consists of a series of PEs arranged in a rectangular grid, as seen in Fig. 1a ("Cartesian" refers to the coordinates that locate each node within the grid). Each of these PEs can take data from the primary inputs and the columns to its left, and usually implements a stateless simple function such as arithmetic addition or logic AND. In order to further simplify the system in terms of multiplexers and routing, the number of inputs available to a certain PE can be constrained to a maximum number of columns to the left (typically one column, in order to avoid excessively large multiplexers).

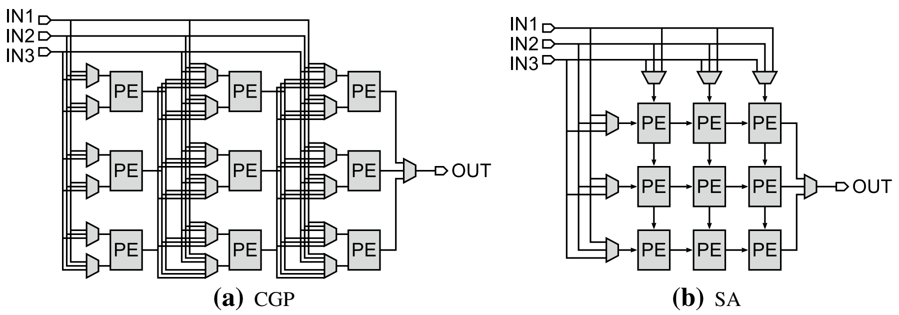

Fig. 1 Example of 3 × 3 CGP and SA topologies with 3 inputs and 1 output. Each PE in these examples has 1 output and 2 inputs, from either the system input or a PE in the previous column. Notice how the interconnections in SA are simpler and there are no multiplexers except for the ones at the input and output of the system

A problem with this topology is that the multiplexers at the input of each PE use a high amount of resources. For example, while an 8-bit adder processing element only needs 8 LUTs in total in modern Xilinx FPGAs, a single 13:1 multiplexer as proposed in [7] (9 inputs + 4 PEs) requires 4 LUTs per output bit [46], 64 LUTs in total for 2 input multiplexers. Therefore, multiplexers alone would represent 89% of the resource usage for this topology.

Another problem with this topology is that the routing of the nets may still be too complex, and that the multiplexers introduce extra delay in the logic path reducing the frequency of operation, although this can be solved by registering the output of the multiplexers in addition to the PE outputs. Furthermore, the complexity of this topology varies with scalability, as larger multiplexers are needed if the array grows vertically.

2.2 Systolic array

An alternative topology which does not suffer this problem is the *systolic array* (SA), first defined in [19] as a generic computing engine, and used as a reconfigurable fabric for implementing EHW in [30, 31]. It was originally intended for complex PE operations but it can be used with simpler PEs as well. Opposite to CGP, inputs to each PE are fixed, connecting each of them to its neighbors (Fig. 1b). The main motivations for choosing this topology in [30] were its use in high throughput processing applications and its suitability for DPR.

The main advantage of this topology is that it removes the need of having a multiplexer at the input of every PE (except at the array input), thus reducing the resource usage of each PE. The reduction in the PE size allows to implement a higher number of them with the same amount of resources.

Additionally, this topology simplifies the routing between processing elements, allowing for shorter data paths and thus lower delay. Its simplicity can also be an advantage in certain *dynamically scalable* EHW systems such as [9].

The disadvantage of this solution is the limited connectivity between PEs, which would force EAs to take longer evolution cycles to obtain correct mappings, or even limiting the maximum quality of the resulting circuits. In other words, there is a tradeoff between resource usage and flexibility: while SA is able to implement a higher number of PEs than CGP for a fixed amount of resources (given the lack of multiplexers), the processing capability that can be obtained from these PEs is limited by the way they are interconnected.

The aim of this work is to analyze this tradeoff in terms of scalability, comparing the performance and resource usage of both SA and CGP architectures with sizes ranging from a few PEs to several hundreds.

3 State of the art of evolvable hardware and FPGA reconfiguration techniques

This section surveys the field of EHW using FPGAs. The focus is set on internal self-reconfiguration rather than on other approaches, explored in the early days of the field, that used external reconfiguration and thus needed support from an external device or PC. First, a brief historical perspective of the limitations imposed by FPGA reconfiguration technology and an important way to overcome them, is included. The text follows with a review of the state of the art of FPGA-based evolvable systems with remarks on the different DPR techniques used. And, finally, the two proposals considered to be the current state of the art are further analyzed to better focus the contents of this work.

EHW in FPGAs has been closely tied to the state of reconfiguration technology and to the possibility of whether *internal reconfiguration* was available to achieve self-reconfiguration, or support for *external reconfiguration* from other device or PC was needed. Xilinx XC6200 FPGA family, in particular the XC6216 device, was used in the considered first EHW attempt with FPGAs [35]. Although missing the important internal reconfiguration feature, this device was once considered the best FPGA-based evolvable platform mainly because: (1) the configuration bitstream was publicly available allowing the creation of efficient representations; and (2) its routing resources were based on multiplexers, enabling random safe modifications of the configuration bitstream.

However, the state of reconfiguration technology changed after this family was discontinued and replaced by Virtex families in the late 90s, posing several restrictions to embrace DPR as a standard practice for self-reconfigurable systems. This turned into a major obstacle for advancements in EHW due to: (1) the MUX-based routing approach being abandoned in favor of a switch-matrix, impeding random, safe bitstream modifications; (2) the lack of adequate DPR tools and design flows at those dates, including the impossibility to relocate modules (partial bitstreams) to different positions of the FPGA; (3) the increasing bitstream sizes and unknown formats; and (4) the insufficient reconfiguration speed to achieve near real-time self-adaptation.

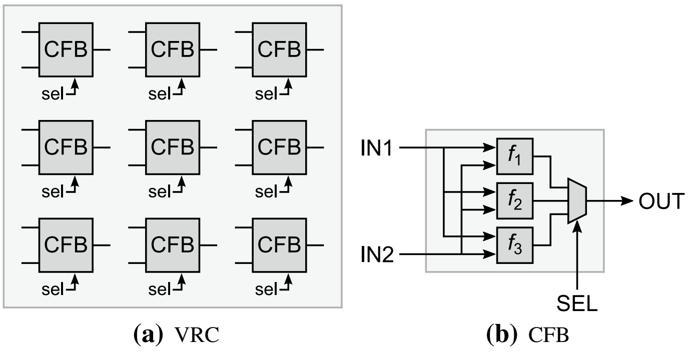

Fig. 2 Structure of a virtual reconfigurable circuit

3.1 Overcoming early DPR limitations: the virtual reconfigurable circuit

A proposal in the first 2000s allowed to keep investigating in the EHW field while DPR technology kept maturing, the *virtual reconfigurable circuit* (VRC) [33]. VRCs were devised as a solution for implementing CGP, thus being VRCs the specific form CGP architectures usually take in FPGA implementations. The CGP topology was inspired by genetic programming and the two-dimensional layout of FPGAs from the previous work of the authors on the evolution of digital circuits, so it has been widely used by EHW practitioners for many different applications [22]. It describes a digital circuit as a directed graph, with a simple integer genotype that describes the functionality and connections of each node of the tree. This genotype is mapped on a kind of grid of computing nodes. Despite its name, CGP usually does not use genetic programming, but a simple *evolution strategy* (ES) [8] with small populations of 1 parent and between 4 and 10 children. Mutation at low rates is the only evolutionary operator since recombination does not seem to affect the search in an effective manner.

A VRC (Fig. 2) is essentially an *overlay* architecture, i.e., a virtual layer that defines an application specific reconfigurable circuit on top of an FPGA fabric to reduce the complexity of the reconfiguration process and increase its speed compared to native reconfiguration. In the specific domain of EHW, a VRC is typically specified using standard HDL descriptions and contains: (1) an array of *configurable functional blocks* (CFBs) acting as the nodes of the array such that each CFB contains all the required functions inside a *processing element* (PE)[1]; (2) a set of functionality multiplexers (one per CFB) to select the required function from the PE; (3) a set of routing multiplexers that provide inter-node connectivity; and (4) a large register as the configuration memory. VRC reconfiguration is very fast given it

[1] PEs might be defined, for instance, at logic gate level (and, or, xor, etc.) to evolve combinational circuits or at the function level (adders, subtracters, shifters, etc.) for other applications such as image/signal processing.

only involves writing this register, which drives the selection signals of all the multiplexers (MUXs). However, large overheads are introduced by this virtual layer: (1) an area overhead due to the simultaneous implementation of every function in every node of the array, plus the MUXs; and (2) a delay overhead due to these MUXs, which also reduces maximum frequency.

3.2 Review of FPGA-based evolvable systems

This subsection contains a survey of works and a discussion on architectural issues regarding evolvable systems in FPGAs using internal (either virtual or native) reconfiguration, with a focus on the different DPR techniques used along the years. Deeper surveys that also consider external reconfiguration in FPGAs, evolution on other devices and extrinsic EHW, i.e., when evolution is performed in a simulator of the target technology, can be found within the relevant literature of the field [15–18, 29, 34, 47].

During the first 2000s Xilinx introduced the *internal configuration access port* (ICAP) [2] which enabled native self-reconfiguration by providing a way to trigger reconfiguration from inside the FPGA. This port, missing in previous MUX-based devices, was a desired step-ahead in order to achieve on-chip, structural circuit evolution. Sadly, reconfiguration speed was at first too slow for EHW applications and, besides, the rest of limiting factors mentioned previously was still present. Altogether, made the use of native reconfiguration in FPGAs being held back for some years, so DPR usage greatly remained in the background within EHW until technology improved and first works began to appear during the mid 2000s. Since Xilinx Zynq devices appeared in the very last years, the new *processor configuration access port* (PCAP) for reconfiguration from the embedded ARM processors was introduced to address the current heterogeneous devices that FPGAs are, offering new possibilities and improved reconfiguration speeds.

Main works are collected in Table 1, which covers EHW dealing with intrinsic evolution in FPGAs either for evolutionary hardware design or for runtime adaptive HW.[2] Whichever the objective is, the following components can always be identified: (1) a reconfigurable area featured by a set of reconfigurable elements determined by the problem domain (3rd column); (2) the EA in charge of driving the evolutionary process (5th column); (3) a HW evaluation module for the fitness computation unit (or k multiple instances for further speed-ups, indicated by '$k \times$ HW') (6th column); and (4) a reconfiguration controller.

[2] These two terms are the main branches in which EHW can be classified. ***Evolutionary hardware design*** is considered as yet another design methodology, based on EAs, that automatically provides a circuit description fulfilling the required specifications; the final solution, usually the fittest circuit, is kept and used to configure/build the final system. By the contrary, ***adaptive hardware*** enables the possibility of hardware self-adaptation in reconfigurable systems. Since the EA has to be embedded in the final system and, along with certain *self/environment-awareness*, autonomous adaptation capabilities (to the inputs, the environment, the specifications and in the presence of faults) are provided so self-adaptive hardware can emerge as long as no human intervention is needed during the adaptation process.

Table 1 FPGA implementations of evolvable digital systems

References	Application	Reconfiguration	FPGA	EA	Fitness
[37]	FIR filters	Register values	Any, not reported	HW	HW
[21]	Image filters	VRC	XC2V3000	HW	HW
[28]	Hash functions	VRC	XC4VFX20	HW	HW
[38]	RBN[a]/Cellular automaton	LUT-DPR (ICAP)	Virtex-II	MicroBlaze	MBlaze
[39]	Image filters	VRC	XC2VP50	PowerPC	HW
[13]	Sonar spectrum classifier	VRC	XC2VP30	PowerPC	HW
[44]	Arith. circuits	VRC	XCV2000E	HW	$2 \times$ HW
[40]	CGP accelerator	VRC	XC2VP50	PowerPC	HW
[10]	Face image classif.	VRC	XC2VP30	MicroBlaze	HW
[43]	Image filters, classif.	VRC	XCV2000E	HW	HW
[42]	MCMs[b]	VRC	XC2VP50	HW	HW
[12]	Face classifiers	SRLUTs[c]	XC2VP	PowerPC/MBlaze	HW
[41]	CGP accelerator	VRC	XC5VFX100T	PowerPC	$4 \times$ HW
[4]	Small comb. circuits	Module&LUT-DPR	Virtex-4	PowerPC	HW
[3]	Classifiers and comb. circ.	Module-DPR	Virtex-4	PowerPC	HW
[1]	Same as [3]	SRLUTs-CFGLUT5[d]	Virtex-5	MicroBlaze	HW
[6]	Image filters	VRC & Module-DPR[e]	Zynq-7000	ARM Cortex-A9	HW
[31]	Image filters	Module-DPR	Virtex-5	MicroBlaze	HW
[11]	Face/sonar classif.	LUT-DPR[f] (ICAP)	Virtex	MicroBlaze	HW
[7]	Image filters	VRC & LUT-DPR (PCAP)	Zynq-7000	ARM Cortex-A9	HW
[5]	Image filters	VRC & LUT-DPR (PCAP)	Zynq-7000	ARM Cortex-A9	$6 \times$ HW
[25]	Image filters	LUT-DPR (ICAP)	Virtex-5	MicroBlaze	$8 \times$ HW
[26]	Image filters	LUT-DPR (ICAP)	Virtex-5	MicroBlaze	$12 \times$ HW

[a]Random Boolean networks

[b]Multiple constant multipliers

[c]Shift register LUTs (various Xilinx FPGAs)

[d]CFGLUT5 is a Xilinx runtime library primitive to change LUTs functionality

[e]DPR using partial bitstream describing functional modules; requires Xilinx module-based methodology.

[f]Lookup table DPR through changing LUT configuration data by direct bitstream manipulation

Typical architectures for FPGA-based evolvable systems closely follows general IP-based designs for FPGAs using a central processor, which holds the EA, and different hard-IP cores, among which an evolvable component is included. Complete hardware evolution experiments in which the EA is also implemented in HW have also been conducted [21, 28, 37, 42–44]. However, these were mostly abandoned given: (1) the advantages for fine tuning the EA using an embedded processor; and (2) the fact that the fitness computation is the most time consuming task, so no significant speed-ups are obtained with a HW implementation of the EA for most applications reported.

Regarding VRCs, different applications have been addressed: image filters [21, 39, 43]; classifiers [10, 13, 43]; cellular automata [38]; combinational logic circuits [44]; hash functions for packet classifiers [28]; and Multiple Constant Multipliers [42], among many others. It is also worth mentioning the work done in HW acceleration of CGP [40], which can be further increased by replicating the fitness unit [41, 44].

3.2.1 FPGA native reconfiguration in EHW

Given some of the issues with DPR being progressively solved by FPGA manufacturers, some works began to appear in the last decade. These can be classified according to the following categories (all using Xilinx FPGAs):

- *Library of pre-synthesized partial bitstreams*: (i) [36] DPR through ICAP of *category detection modules* (CDM) with different number of *functional units* (FU) per CDM for previous VRC-based classifiers [13]; (ii) [4] DPR through ICAP to change the functionality of nodes (or columns of nodes) of VRC-CGP-like structures with static routing for small combinational circuits and [3] for more complex circuits and classifiers; (iii) [31] DPR of the PEs of an evolvable SA using an optimised ICAP controller with relocation capabilities and a self-healing mechanism for image filters.
- *LUT reconfiguration through shift register functionality (SRLUTs)*, changes LUT functions by directly shifting configuration bits into them[3]: (i) [12] online incremental evolution of FUs in the CDMs to reduce area and reconfiguration time of evolvable classifiers (of previous ICAP [36] and VRC [13] versions); (ii) [1] low-level evolution of nodes from [4].
- *DPR-based LUT reconfiguration through direct bitstream manipulation*: (i) [38] evolvable Random Boolean Network with arbitrary connectionism by using LUTs as multiplexers while keeping a fixed routing; (ii) [11] fine-grained LUT reconfiguration in evolvable classifiers (updates previous architecture [12] to support fine-grained partial reconfiguration of LUTs); (iii) [4] low-level evolution of node functionality and routing of VRC-CGP-like structures; (iv) [25] optimised PE implementation in evolvable SAs, improving area cost and maximum frequency from [31].
- *Hybrid VRC-DPR methods* for CGP-based image filters using PCAP. Virtual reconfiguration using multiplexers (but adding flip-flops at the outputs to reduce delay impact) is used for PEs interconnection, while PE reconfiguration is done

[3] A serious drawback of this reconfiguration strategy is the decreasing number of LUTs with this operating mode: 100% in Virtex-II Pro and around 25% in Virtex-5.

through: (i) [6] DPR using pre-synthesized partial bitstreams; (ii) [7] DPR-based LUT reconfiguration for approximate PEs by direct bitstream manipulation.

3.3 Comments on the current state of the art in FPGA-based EHW systems

The previous analysis have introduced the main works done in EHW using FPGAs from the first 2000s. This last subsection further analyzes the two sets of works that can be considered the current state of the art. Both of them try to address typical VRC-CGP overheads, i.e., area and delay. Image filtering for noise removal (using PEs at the function level, such as adders, subtracters, etc.) is the selected proof of concept application in both cases. There is another architectural overhead derived from CGP architectures in FPGAs, the **implementation scalability**,[4] which has not been given the importance it probably deserves from the point of view of the authors of this work, at least for runtime adaptive hardware; hence it is also included herein as a key factor to be considered for the future possibilities of the EHW field. The reason to consider the implementation scalability as a *new* overhead to be addressed is mainly due to the implications it has both (1) on fault tolerance and self-healing and (2) on the maximum achievable complexity of the evolved solutions, which might be another way to address the *scalability of representation* issue in EHW [14].

The two works mentioned previously as the current state of the art are:

- An architecture based on traditional SAs together with a reconfiguration engine with relocation capabilities [27]. PE reconfiguration is addressed by both using DPR with a library of partial bitstreams [29, 31] and DPR-based LUT reconfiguration [25]. Area is reduced using the SA architecture by only having one PE per node at the same time in the FPGA while delay is improved by restricting PE interconnection to the four closest neighbors and thus eliminating routing multiplexers. This favors local, short interconnections that further reduce propagation delay and improve dynamic runtime scalability, both in the number of SAs [25] and in the number of PEs per SA [9].
- A hybrid VRC-DPR approach [6, 7] for CGP structures that keep virtual reconfiguration using multiplexers for interconnections and both DPR using a library of partial bitstreams [6] and DPR-based LUT reconfiguration [7] for PE reconfiguration. Area is reduced in this case by eliminating the functional multiplexers coming from the simultaneous implementation of all the PEs in every node of the array. Besides, delay is favored given the elimination of the functional multiplexers. However, the routing multiplexers from the CGP architecture are maintained, which keep penalizing delay and architecture scalability.

[4] This is *scalability* from an architectural point of view, which considers both the maximum size of the evolvable processing structures (CGP, SA, etc.) and the maximum number of PEs, for a given set of FPGA resources. Not to be confused with *scalability of representation* (related to the size of circuits that can be evolved in a reasonable time considering the chromosome representation) and *scalability of evaluation* (related to how to reduce evaluation time).

The most up-to-date reports from these works, the SA [26], and the hybrid VRC-DPR approach [7], are further analyzed considering area and maximum operating frequency achieved, thus evolution speed. Regarding area: the SA [26] requires 16 LUTs and 8 flip-flops for each PE (100% of the available reserved logic resources for each PE is used so no further reduction is possible) in Virtex-5 devices; the hybrid VRC-DPR approach [7] requires 8 LUTs and 8 flip-flops per PE for a set of *approximate* PEs in a Zynq-7000 device. Regarding speed: the SA [25, 26] works at up to 500 MHz, the fastest work reported so far to the best of these authors knowledge, achieving 19,000 circuit reconfigurations and evaluations per second on a system with a single array or up to 139,000 with 12 arrays in parallel[5]; the hybrid VRC-DPR approach [7] works at 300 MHz, which results in 8700 evaluations per second or up to 30,000 on 6 arrays.

Before concluding the survey of the field, it is worth mentioning here that the implementation scalability is precisely the main topic addressed in this work: i.e., how well (in terms of area and delay) do SA and CGP architectures compare when scalability at the number of PEs is considered.

4 Systolic array system description and evaluation

In order to evaluate and compare the SA and CGP topologies, a case study has been developed: an evolvable image filter based on a 3×3 pixel *sliding window*, which will be used for removing noise from a 128×128 pixel grayscale image. This section details the SA implementation (the CGP counterpart will be discussed in Sect. 5), and additionally describes the PE architecture that will be used in this case study for both SA and CGP implementations. The SA version of the filter has been implemented on a Xilinx Virtex-5 LX110T FPGA, which is later used to perform the tests, relying on DPR of LUTs through the ICAP of this FPGA to change the functionality of PEs as described in [26].

The system feeds one 3×3 window per clock cycle to an SA with each PE implementing a specific function. The window is centered on the pixel to be filtered, starting with the pixel at the top left corner of the image and shifting the window one pixel to the right per clock cycle. When the window reaches the right end of the image, it continues with the pixels on the left side of the image one line below. Left and right edges are treated as if the rightmost pixel in one line were adjacent to the leftmost pixel in the line below, and top and bottom edges are treated as if they were adjacent to black pixels (value of 0). This approach may result in the edges having worse quality than the rest of the image, but simplifies the implementation and avoids removing a one pixel border from the output image.

[5] Since Virtex-5 devices feature around 60% smaller reconfiguration frames (which is the minimum reconfigurable unit) compared to Zynq-7000, an implementation in this platform would probably increase the reconfiguration time of the SA. This means the 19,000 evaluations reported [26] would be fewer in a Zynq device, but in any case probably well over the 8700 figure.

Table 2 Functions implemented by the PEs

Index	Equation	Description
0	$N + W \bmod 256$	Addition (modulo 256)
1	$N + N \bmod 256$	Multiply by 2 (modulo 256)
2	$W + W \bmod 256$	Multiply by 2 (modulo 256)
3	$\min(N + W, 255)$	Addition (limited at 255)
4	$\min(N + N, 255)$	Multiply by 2 (limited at 255)
5	$\min(W + W, 255)$	Multiply by 2 (limited at 255)
6	$\left\lfloor \frac{N+W}{2} \right\rfloor$	Average (rounded down)
7	255	Constant value of 255
8	$\left\lfloor \frac{N}{2} \right\rfloor$	Divide by 2 (rounded down)
9	$\left\lfloor \frac{W}{2} \right\rfloor$	Divide by 2 (rounded down)
10	N	Identity (pass through)
11	W	Identity (pass through)
12	$\max(N, W)$	Maximum
13	$\min(N, W)$	Minimum
14	$\max(N - W, 0)$	Subtraction (limited at 0)
15	$\max(W - N, 0)$	Subtraction (limited at 0)

The SA configuration, i.e., the function implemented by each PE and the window pixel that is fed to each SA input, is the same for the whole image. The selection of the optimal configuration is chosen using an EA. In order to evaluate the quality of a specific SA configuration (the *fitness* of the solution), the output image can be compared pixel by pixel with a reference image using the *sum of absolute errors* (SAE), defined as

$$\mathrm{SAE} = \sum_{i=1}^{128} \sum_{j=1}^{128} \left| \mathrm{output}_{ij} - \mathrm{reference}_{ij} \right|$$

This fitness criterion is chosen instead of others such as the mean squared error or the PSNR because it can be easily implemented in hardware, and is still able to drive the EA.

4.1 Systolic array

The filter is based on an SA with a size of 24 × 24 PEs, which is able to emulate smaller sizes by leaving some of its PEs unused. These PEs are implemented by directly instantiating LUTs with fixed positions on the FPGA, which allows changing their functionality by modifying the LUT configuration through DPR, as described in previous work [25, 26].

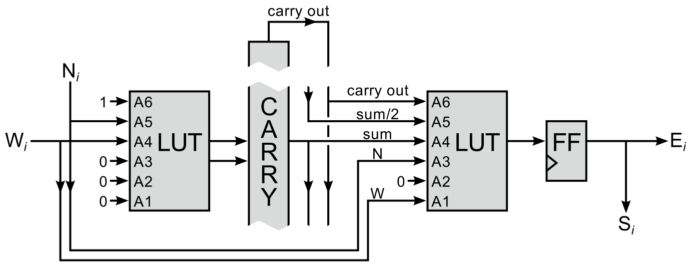

Fig. 3 Fragment of a PE. Each PE is constituted by 8 fragments like this, one per bit. The *sum/2* signal is the *sum* signal (extended with the *carry out* bit) shifted one bit to the right

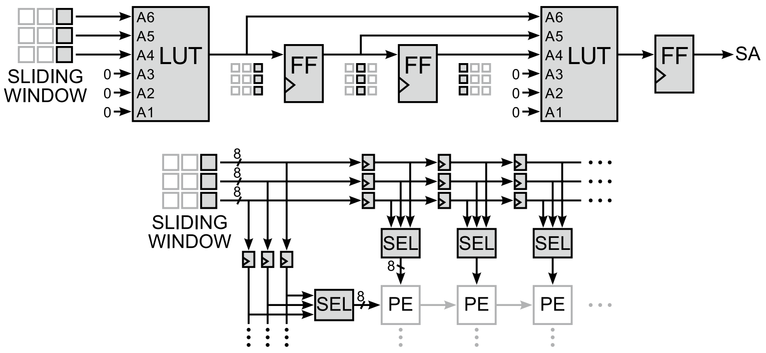

Fig. 4 Top: fragment of an input selector; each input selector is composed by 8 fragments like this. Bottom: pipelining of the input selectors

4.1.1 PE description and structure

Each PE has two 8-bit inputs connected to the adjacent elements situated above (“north”, *N*) and to its left (“west”, *W*), and one output connected to the PEs below (“south”, *S*) and to the right (“east”, *E*); and implements a simple function such as identity, addition (modulo 256 or saturating at 255), subtraction, constant value of 255, etc. The list of available functions is shown in Table 2.

The structure of a PE is shown in Fig. 3. Each PE is composed of 2 stages, one for calculating a sum and carry derived from the inputs and one for generating a result based on the inputs, sum, and carry. The first stage can also be used for multiplying one input by 2 (by adding it to itself) or comparing the inputs. In total, each PE uses 16 LUTs, 8 FFs, and two 4-bit carry chains, all of which fits into 4 CLB *slices* on a Xilinx Virtex-5 FPGA [45]. The carry chains allow to efficiently implement arithmetic functions, and the FFs register the result in order to obtain a pipelined architecture. This structure is able to implement all of the functions in Table 2 by just changing the content of the LUTs. The order of PE inputs has been chosen in order to optimize the circuit speed, and the unused inputs have been tied to logic 0 in order to reduce the amount of data to be reconfigured.

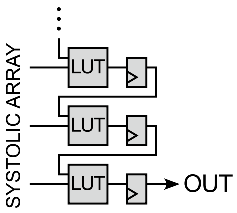

Fig. 5 Diagram of the output selector (for 1 of the 8 bits). The size of this selector depends on the height of the array

4.1.2 Input and output selectors

PEs on the north and west borders of the SA take their input from one of the pixels of the 3×3 input window. The pixel is selected by an *input selector*, which is a circuit implemented using LUTs with a functionality similar to a multiplexer, but relying on reconfiguration of the LUT content rather than a "select" line to change the input passed to the output. The structure of an input selector is shown in Fig. 4, and is similar to that of a PE. In order to simplify routing and reduce the number of LUTs, only 3 pixels of the window are input to the selector; the other 6 are obtained by delaying the input.

The current implementation does not utilize all LUT inputs, and could be extended to windows of a size of up to 6×6 pixels without adding extra LUTs. The input selectors are cascaded and registered in between for consistency with the internal pipelining of the array; the need for doing this is discussed in Sect. 4.3.

A whole input selector uses 16 LUTs (2 per bit) and 48 FFs (6 per bit, including the 3 used for pipelining), and its functionality is changed by reconfiguring the content of its LUTs.

Rather than always using the bottom right PE as the SA output, the output can be selected from any of the PEs on the rightmost column, similarly to how it is done in CGP. This has little impact on the resource usage of the array and in exchange gives it more flexibility during the evolution; additionally, it prevents a specific PE chosen as output during the design stage from becoming a single point of failure [32]. The *output selector* is depicted in Fig. 5, and it uses 8 LUTs and 8 FFs per SA output. This selector is pipelined for both scalability and consistency with the rest of the SA. Fault tolerance techniques such as *triple modular redundancy* could be applied to this selector so that it does not become a new single point of failure, but this is out of the scope of this work.

4.1.3 Resource usage

The whole 24×24 array uses 10,176 LUTs and 7680 FFs/latches. Smaller arrays could be implemented using fewer resources, for example a 10×10 array would only require 2000 LUTs and 1940 FFs/latches. LUT usage for other sizes can be calculated as $N_{\mathrm{LUTs}}(H \times W) = 16 \times H \times W + 16 \times (H + W) + 8 \times H$ (where H and W are the array height and width in PEs). Figure 6 shows LUT usage for different

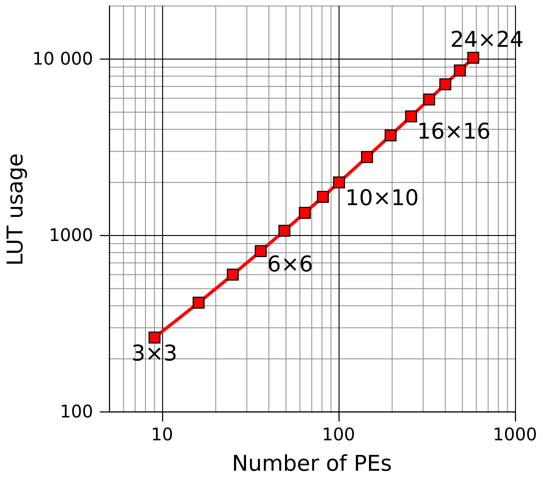

Fig. 6 LUT usage for different sized SAs

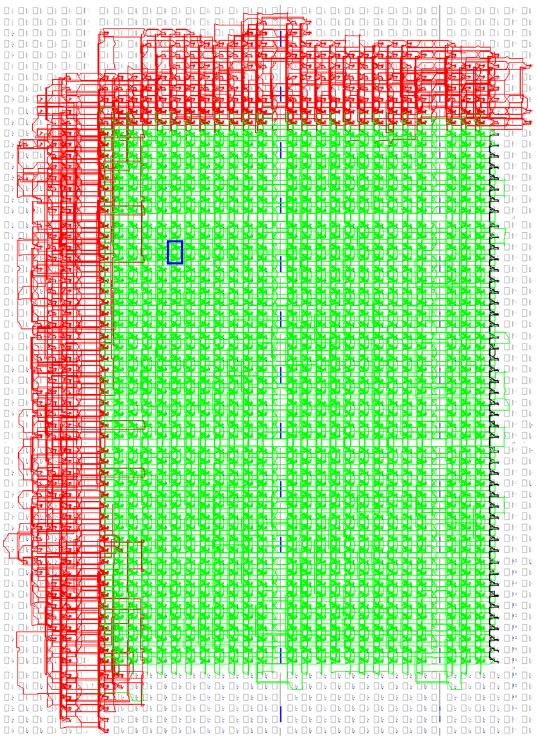

Fig. 7 Screenshot of Xilinx FPGA Editor showing the 24 × 24 array. Red: input selectors, including pipelining. Green: 24 × 24 PEs, with an individual PE marked with a blue box. Black: output selector (Color figure online)

sizes: as it can be seen, the relationship quickly becomes linear (despite of the overhead introduced by the input and output selectors), which is beneficial to scalability.

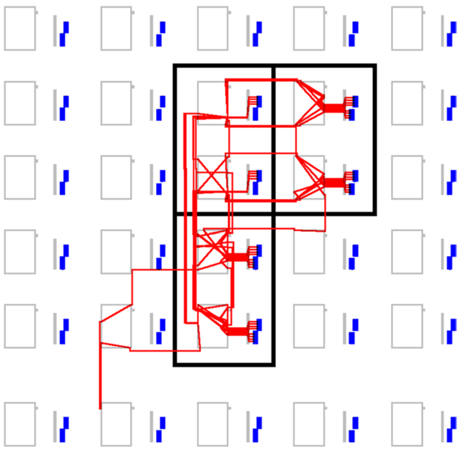

Fig. 8 Detail of the interconnections between a PE (top left) and its two neighbors connected to its output (right and bottom). Despite the array size, the interconnections between neighbors are kept short

Due to the small number of interconnections between PEs, the whole array can be implemented in a very compact size without routing problems, as seen in Fig. 7, where the area in green has a LUT usage of 100%. Since each PE in an SA connects only to its neighbors, the interconnections are very short (Fig. 8). This results in a reduced net delay, making the system able to achieve a theoretical frequency of 340 MHz (on a Virtex-5 LX110T with speed grade − 1) but experimentally proven to reach up to 400 MHz without showing any faults derived from timing errors.[6]

4.2 Evolutionary algorithm

The algorithm used to obtain an optimal configuration is based on a simplified genetic algorithm, where each gene is an integer value that represents the function of a PE (from 0 to 15), the window pixel selected by an input selector (from 0 to 8), or the output selected by the output selector (from 0 to height − 1).

The EA starts initializing the systolic array so that all input selectors select the central pixel of the 3×3 window, all PEs copy their west input, and the output selector selects the output from the bottom right PE, in order to obtain an identity filter as the initial configuration. This filter is evaluated by filtering a training image and comparing the filter output (currently identical to its input) with a reference image, obtaining the filter's *fitness* as the SAE between the output and the reference.

After initialization, K genes are selected randomly with a uniform distribution, and their value is changed randomly (within the gene's range of possible values). The newly obtained configuration is evaluated and compared with the previous one. If the fitness of the new configuration is worse (higher) than the previous one, the changes are reverted. Otherwise, the changes are preserved. In this work, a *mutation rate* of $K = 3$ has been used because it has experimentally shown good results

[6] Smaller implementations of up to 16×16 PEs have been proven to reach 500 MHz without showing timing errors.

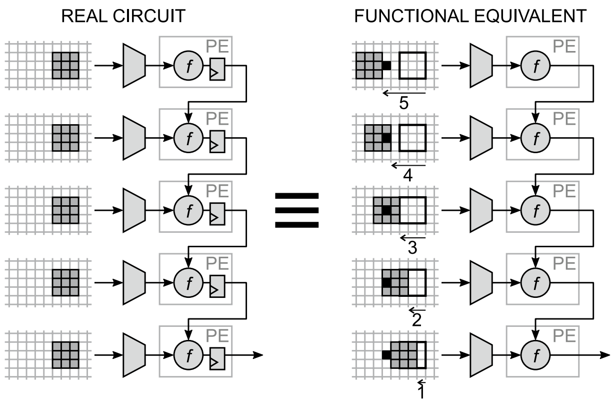

Fig. 9 Left: 5×1 SA with inputs directly connected to the PEs, each of which registers its output. Right: this delay is equivalent to shifting the input window one pixel per clock cycle to the left; as a result, each effective input window is positioned differently with respect to the pixel marked in black

regardless of the size of the SA; the analysis of the optimal K is however out of the scope of this work. This process is repeated for a certain number of iterations. This EA is sometimes referred to as (1 + 1)-EA, since it has a population of 1 *parent* and an offspring of 1 *child*.

This type of EA is prone to getting stuck at local optima. In order to reduce the risk of this happening and improve the efficiency of the EA, 8 separate evolutionary runs are performed concurrently. Every 2048 iterations, all runs are compared; the one with the worst fitness is terminated and the one with the best is forked. This promotes good evolutions in addition to reducing the effect of bad evolutions that have gotten stuck. This strategy was used in [25], and allows obtaining similar results to the single run evolution in 4 times fewer evaluations; in addition, it can be easily parallelized.

The EA is run for 32,768 iterations, evaluating a total of 262,144 candidate configurations.[7] Once finished, the best result from the 8 evolutions is selected as the solution.

4.3 Is it necessary to pipeline the inputs?

Typically, pipelined architectures require data to follow a strict registering scheme so that data from different clock cycles do not mix. However, this is not a hard requirement in the case of EHW, specially when signals from different clock cycles are related, since the EA can find solutions which take advantage of this signal mixing. Therefore, the extra registers needed to correct the pipelining could be removed if this simplifies the design without affecting the results.

[7] This number of evaluations (2^{18}) was chosen experimentally as a good tradeoff between filter quality and evolution time, with significantly better results than 2^{17}, whereas 2^{19} was only marginally better [25]. Powers of 2 were used because they were more practical for parallelizing and partitioning the EA.

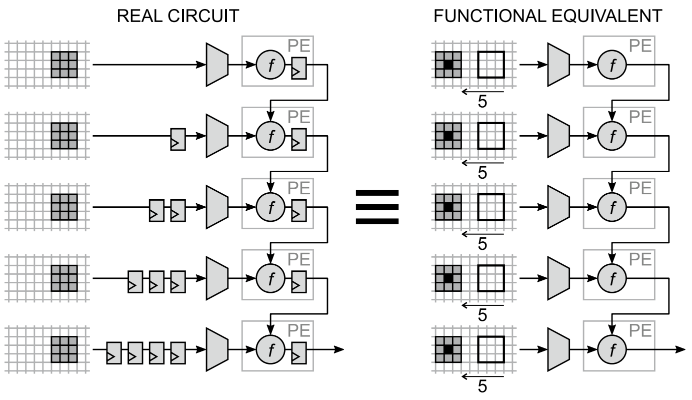

Fig. 10 Left: 5 × 1 SA with correctly pipelined inputs. Right: the delay affects all input windows equally, so now all effective input windows have the same position with respect to the pixel marked in black

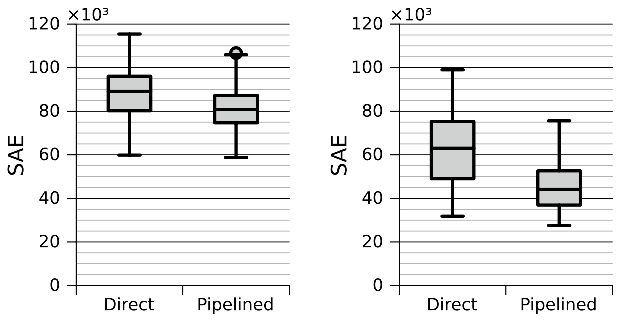

Fig. 11 Fitness (SAE, lower is better) obtained after evolving 4 × 4 (left) and 8 × 8 (right) SAs, showing the effect of pipelining the inputs and outputs instead of connecting them directly. Results after repeating the evolution 100 times with different initial random seeds

In this work, pipelining inputs and outputs in addition to the PEs is done in order to ensure that data from different windows propagates separately through the array as different *wavefronts* [20], so that the array behaves functionally as if it were purely combinatorial.

Figure 9 shows a 5 × 1 SA whose inputs are directly connected to the primary input (3 × 3 pixel window). However, since each PE registers its output, the propagation latency from each input to the output varies, being of 1 clock cycle for the bottom input but 5 clock cycles for the top one. This is functionally equivalent to having a purely combinatorial SA with added latencies at each input. Since the input window moves one pixel per clock cycle to the right, this added latency is equivalent to shifting the input window to the left by a varying amount of pixels. Therefore, the center of the window at the central input (marked in black in Fig. 9) will not match the center of the window at the rest of inputs.

This discrepancy in the latency can be corrected by adding extra registers at the inputs, as shown in Fig. 10. Here, the input-output latency is 5 clock cycles for all

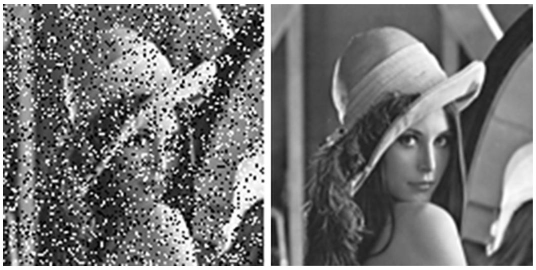
Fig. 12 Training image with 20% noise (left) and reference image without noise (right) used in the evolution

the inputs, and therefore the circuit behaves as if each input window has been shifted 5 pixels to the left, making all the input windows have the same center. In other words, this is equivalent to a purely combinatorial SA with an added latency of 5 clock cycles.

Nevertheless, previous work [9, 24, 30, 31] has shown good results without pipelining the inputs. However, this presents several disadvantages:

- It makes the array latency uncertain and unpredictable, since it will be a combination of multiple latencies, each with a different impact on the result depending on the SA configuration. On [31], this is solved by comparing the output with differently shifted versions of the reference by using multiple comparators and reporting the best one.
- It makes certain inputs take a window that is too far from the “pixel of interest” (marked in black in Fig. 9) rather than centered around it, which may make those inputs less useful. This effect can be seen as an advantage since it increases the effective window size, but as Fig. 11 shows it results in a worse performance.
- It makes the system harder to characterize and simulate.

The effect of not pipelining the input is greater the larger the array is, since the variability of the latency grows with the size of the array. Figure 11 shows the result of evolving arrays of size 4×4 and 8×8 without and with pipelining. As can be seen, the effect of pipelining the inputs is small for a 4×4 array (as used in [30]), but for an 8×8 array it noticeably improves the results.

4.4 Experimental results

The system described above has been tested for different array sizes in order to analyze its scalability. The tests have been performed on a *Lena* image with a 20% of *salt and pepper*[8] noise (Fig. 12), using it as the training input to the EA described in Sect. 4.2 and a noise free version as a training reference with which to calculate the SAE in order to obtain the fitness. A total of 100 tests have been run for each size,

[8] *Salt and pepper* noise consists in replacing randomly selected pixels (in this case, 20% of them) with black or white pixels.

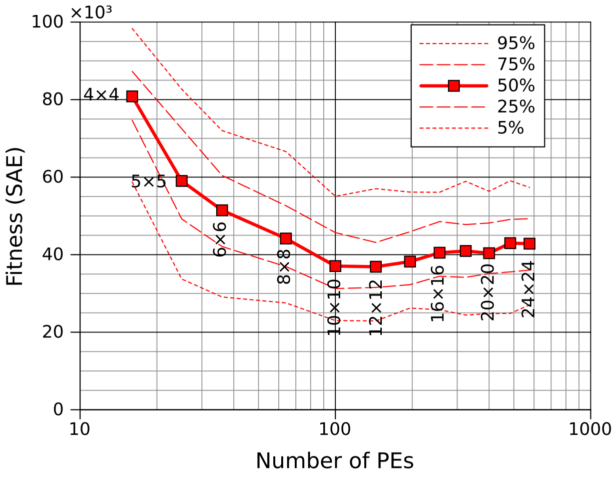

Fig. 13 Fitness (SAE; lower is better) after evolving SA-based filters of different sizes. The solid line represents the median of the 100 evolutions for each size, and the dashed lines represent different percentiles

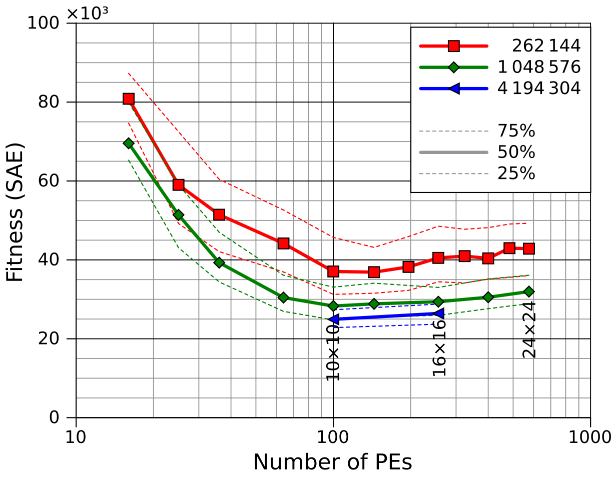

Fig. 14 Results obtained with a four times longer evolution, comparing the former length of 262,144 evaluations with an extended length of 1,048,576. Two points of a 16 times longer evolution (4,194,304 evaluations) are shown as well

each with a different initial seed for the random number generator. Figure 13 shows the resulting fitness for each size (median and different percentiles). As a reference, the original image has an SAE with respect to the reference of 416,297, and the image filtered with a *median filter* using a 3×3 pixel window has an SAE of 84,187.

As can be seen in Fig. 13, the larger the SA, the better the result; up to an optimum size of 10×10. After this, the results stop improving, and even get slightly

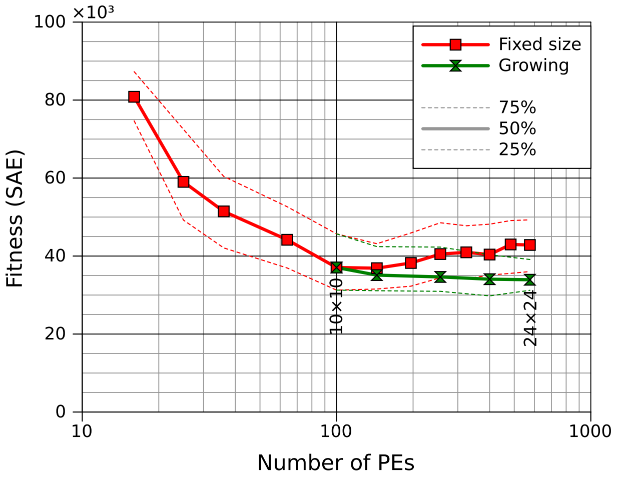

Fig. 15 Evolution results growing from an initial SA size of 10 × 10 to a specific size, compared with the previous results obtained with a fixed size

worse, which might be counter-intuitive since large arrays can implement all the functions of small arrays. This might be due to the *search space* becoming excessively large and the current EA not being able to find good solutions in such a broad space, getting easily stuck at local optima. Leaving the EA for a longer time helps finding better solutions, but as Fig. 14 shows, this benefits small arrays as well, and the same optimum size of 10 × 10 is still observed, with larger sizes resulting in slightly worse results.

A possible way to overcome this problem is to modify the EA so that, instead of evolving the whole array, it starts evolving a 10 × 10 PE fragment of it, growing progressively during evolution. This allows finding a solution for a small size and then improving it progressively as more resources become available. However, as shown in Fig. 15 (for the same total length of evolution), results barely improve with respect to the 10 × 10 filter.

4.5 Conclusions

The described SA architecture can be implemented in a very compact size, which results in a very small resource usage and good timing behavior. A 24 × 24 SA uses 10,176 LUTs (not including the control logic), less than 15% of the LUTs available on a Xilinx Virtex-5 LX110T FPGA, and a 10 × 10 one can be implemented in 2000 LUTs (less than 3%), which would allow to put several SAs in parallel as is done in [25, 26].

The performance of SAs scales well with sizes of up to 10 × 10 PEs. This size is considerably large compared to previous implementations of 4 × 4 PEs [31] which already yielded relatively good results. However, the use of arrays larger than 10 × 10 did not provide a significant improvement. This could be due to limitations of either the EA or the topology itself. Nevertheless, the small size of this architecture makes it suited for integration with larger systems, or in FPGAs where the available resources are scarce.

It is important to correctly pipeline the inputs and outputs of the SA in order to be synchronous with the PEs, specially for large sizes, since even if the EA is able to find good solutions without the pipelining correction (this is, with the inputs and outputs directly connected to the PEs), this correction provides better results.

Notice that, although the system has been trained with a single image, previous work [31] shows that the filters resulting from this kind of system can be used with different images, i.e., the obtained filter is not specific to one image but is generalizable to other images with the same kind of noise. Additionally, these systems adapt well to different types of noise, and can be trained even without a noise-free reference [24].

5 Comparison with CGP

An alternative to the SA architecture that is widely (and almost uniquely) used for evolvable hardware is the CGP architecture. As described in Sect. 2, this architecture is similar to the SA but with each PE having multiplexers at its inputs that allow to select them from either the primary inputs of the array or from the output of one of the PEs from one of the columns to the left, rather than having the inputs connected directly to other neighboring PEs.

The removal of this restriction in the PE interconnection leads to an increased flexibility in the filters that can be implemented with the same number of PEs with respect to the SA. However, CGP has some disadvantages over SA:

- The multiplexers at the input of the PEs introduce a severe overhead in resource usage. Additionally, they can affect the maximum operating frequency since they add extra logic levels between registers, although this can be easily solved by registering the output of the multiplexers in addition to the PE outputs (which conversely results in an extra FF usage).
- The routing of the design becomes more complex, since a PE output is connected to a great number of PEs, so the interconnections will likely be longer than the ones shown in Fig. 8, resulting in a worse timing.
- The vertical scalability of the array is more complex to achieve. The homogeneity of an SA makes the structure of each PE independent of the size of the array; however, the size of multiplexers in CGP depends on the array size, having more inputs for larger sizes. This makes the complexity of each PE grow with the size of the array.

In [7], a CGP-based evolvable hardware system is implemented using a hybrid approach in which the PE functionality is changed using DPR, but the inputs are switched using actual multiplexers. This implementation uses a simplified set of PE functions that fits in 8 LUTs; however, this does not account for the LUTs needed to implement the PE input multiplexers.

Table 3 Functions implemented by the PEs

Index	Equation	Description
0	255	Constant value of 255
1	$a + b \bmod 256$	Addition (modulo 256)
2	$\min(a + b, 255)$	Addition (limited at 255)
3	$\max(a - b, 0)$	Subtraction (limited at 0)
4	$\left\lfloor \frac{a+b}{2} \right\rfloor$	Average (rounded down)
5	$\left\lfloor \frac{a}{2} \right\rfloor$	Divide by 2 (rounded down)
6	$\max(a, b)$	Maximum
7	$\min(a, b)$	Minimum

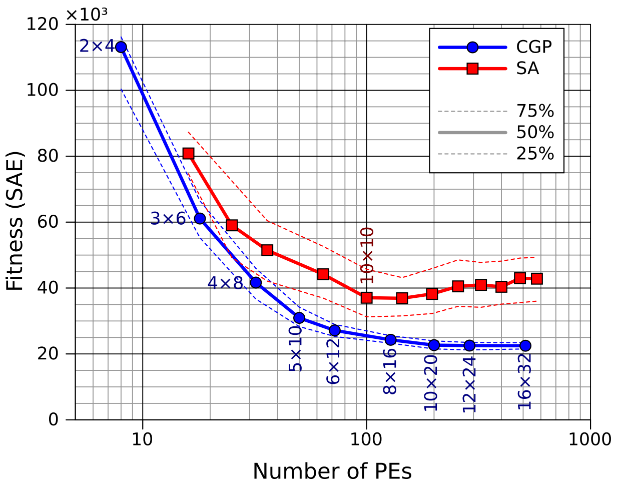

Fig. 16 Fitness (SAE; lower is better) for different sizes of SA and CGP

5.1 Description of the CGP implementation

In order to compare SA and CGP, a similar implementation of the evolvable image filter using CGP instead of SA has been evaluated in a simulated computer model. The hardware implementation modeled in this simulation is described as follows:

- The size of the CGP circuit always has a ratio of 1:2, since this ratio seems to be common in CGP implementations [6, 7, 39]. This is, there are twice as many columns as PEs per column.
- The PE structure is identical to the one for SA described in Sect. 4.1, with two inputs (*a* and *b*) and one output.
- Each PE input is connected to either a pixel on the *primary input* (3×3 pixel window) or the output of one of the PEs from the column immediately to the left. This connection is selected using two multiplexers, one for each PE input.

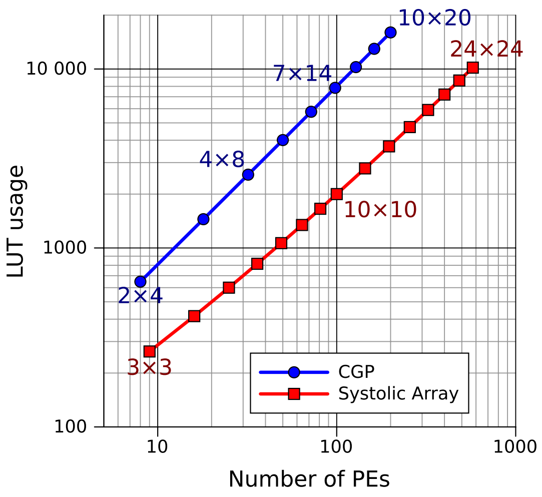

Fig. 17 Comparison of resource usage versus number of PEs for both SA and CGP

- Each PE can implement any of the functions in Table 3. This is a subset of Table 2; the missing functions can be obtained by setting both PE inputs to the same source; for example the identity function can be implemented as the maximum of one value and itself. This may have the effect of reducing the search space so that the evolution converges faster.
- The output of the array is selected from one of the PEs on the last stage.
- Inputs are pipelined the same way as in the SA to make them behave synchronously with the rest of the array. The output needs no extra pipelining since it always comes from the last stage.
- All other aspects of the system (sliding window, fitness criterion, training image set, EA...) are identical to the ones for the SA-based implementation described in Sect. 4.

5.2 Experimental results

The described CGP system has been modeled in a computer simulation and evaluated using the same EA as the SA, with each gene representing a PE function (from 0 to 7), a PE input (from 0 to 8 in the first stage and from 0 to height + 8 in the rest), or the selected output (from 0 to height − 1); therefore, each PE is described by three genes. As can be seen in Fig. 16, CGP outperforms an SA with the same amount of PEs, and its performance gets saturated at a lower (better) fitness value, at a size of 10×20 PEs. This is no surprise since, as it was stated, CGP has a higher flexibility than SA, and is therefore able to obtain better solutions.

However, although this comparison may make sense from the point of view of the computational complexity of the resulting filter, it is not so much the case when bringing into consideration the results in terms of the resources needed for an FPGA implementation.

As it was mentioned, the PEs in CGP use more resources than in SA due to the input multiplexers. In particular, if a PE can get either of its inputs from 9 primary

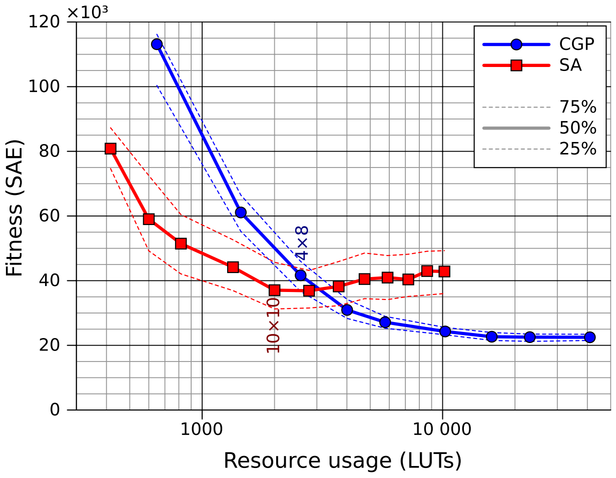

Fig. 18 Fitness for different sizes of SA and CGP, using LUT usage rather than number of PEs as the horizontal axis

inputs and up to 7 PEs from the previous stage, it would need an 8-bit 16:1 multiplexer on each of the two inputs to select among these 16 possibilities. According to [45], 4 LUTs are needed to build a one-bit 16:1 multiplexer; therefore, a total of $4 \times 8 \times 2 = 64$ LUTs are needed only for the multiplexers. Adding the 16 LUTs needed to implement the PE function, this makes a total of 80 LUTs per PE, which would mean that CGP uses 5 times more LUTs than SA (not counting the input and output selectors of the latter, which represent a small fraction of the resources for large arrays). Thus, the formula for the LUT usage of CGP is $N_{\mathrm{LUTs}}(H \times W) = 80 \times H \times W$ (neglecting the output multiplexer).

Figure 17 shows a comparison in LUT usage for both SA and CGP. As can be seen, an SA of 10×10 PEs can implement 4 times more PEs with the same number of LUTs as CGP (including input and output selectors), or from a different point of view, needs only a 25% of the resources needed to implement CGP with a similar amount of PEs.

Using LUTs as a comparison criterion rather than PEs, Fig. 18 is obtained. It can be observed that, although CGP eventually outperforms SA for large sizes (of 5×10 PEs or more), results for SA are better than for CGP for resource usage values below 3000 LUTs.

This reflects the main advantage of SA over CGP: its reduced resource usage allows implementing a higher number of PEs, resulting in higher computational capabilities with respect to CGP for the same cost. This is specially beneficial for implementations that seek to minimize the resource usage, since the performance of a 4×8 CGP architecture matches that of a 9×9 SA, which can be implemented in 35% fewer LUT resources.

Although the simpler interconnectivity of SA limits its scalability, making CGP perform better for very large resource usages, the point at which CGP surpasses SA in terms of performance per resource usage is unusual to reach, since typical

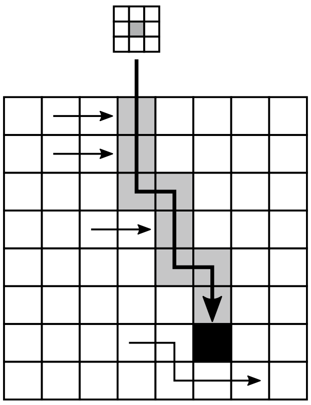

Fig. 19 One possible way of routing a primary input to an arbitrary PE (marked in black). This requires sacrificing several PEs (marked in gray) and blocks other possible data paths (represented as small arrows)

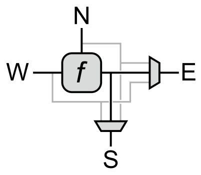

Fig. 20 Functional diagram of the bypassable PE. The multiplexers represented here are actually integrated into the second stage LUT, as shown in Fig. 21

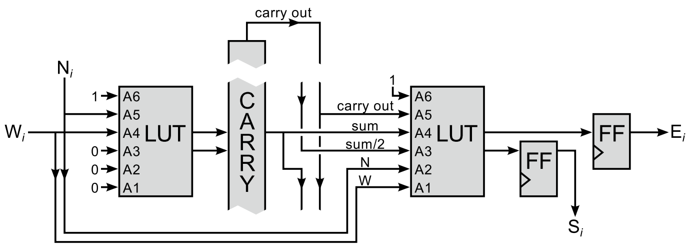

Fig. 21 Architecture of the new PEs. The A6 input of the second LUT is tied to 1 as in the first one, in order to obtain a 2-output LUT. Both outputs are registered separately. The functionality of the resulting PE is a superset of the old one, so all the previous functions can still be implemented

implementations consist of CGP sizes of 4×8 [5, 7] or SA sizes from 4×4 to 8×8 [9, 25, 30], all of them below this point.

6 Improving the connectivity of SA

A possible reason why CGP outperforms SA for large sizes is the ability of all PEs to get the primary input in addition to the input from neighbor PEs. In an SA, on the other hand, a PE needing the primary input would only be able to get it if said input were propagated through other PEs configured with identity functions (Fig. 19).

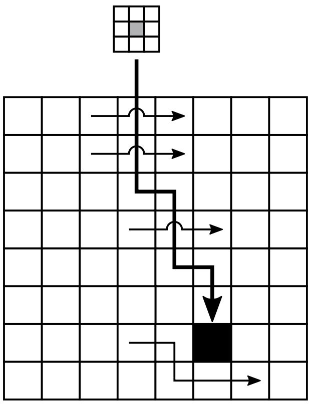

Fig. 22 With the new PEs, routing a primary input to an arbitrary PE no longer blocks other possible data paths, which are now able to cross to the other side, nor does it require sacrificing several PEs for a single PE to be able to access the primary input

This chain of pass-through PEs would not only waste resources, but also act as a barrier preventing other data paths to cross it.

In order to address this problem, the PEs have been modified so that they can be *bypassed* as depicted in Fig. 20: rather than having both outputs take the same value, each output can be configured independently to take either the result of the PE function or one of the inputs. This allows a PE to be configured as a “bridge” that copies *N* to *S* and *W* to *E*, or to repeat one of its inputs (which would otherwise be blocked by the PE itself) to a downstream PE.[9]

Unlike the multiplexers needed by CGP PEs, the simplicity of these—which only require selecting between three possible values—allows them to be integrated directly into the second stage of the current PE architecture as seen in Fig. 21, and thus *they do not increase the LUT usage at all.* However, this change does increase the FF usage as both outputs need to be registered separately, although this is not a problem since half of the FFs in the SA area were not being used; furthermore, new FPGA families such as the 7 Series have twice as many FFs per slice as the Virtex-5.

The aim of this bypass feature is to be able to transmit the primary input to a certain PE without acting as a data barrier or sacrificing PEs (Fig. 22). It should be noted that PEs used for this purpose have their functionality constrained, as one of their inputs is restricted to the primary input to be passed. However, this approach also has advantages over the described CGP architecture, since a PE can not only take its input from a primary input but also from a distant PE, which in the current CGP model would require using PEs as pass-through functions.[10]

Since these PEs already have a built-in pass-through functionality, functions 10 and 11 from Table 2 are not used, reducing the total number of functions to 14. The functionality of the two multiplexers is mutated independently from the PE function,

[9] This behavior is close to the one originally proposed for SA in [19], where a PE could either repeat its input or act in a cascaded manner. Nevertheless, its application to a DPR-based EHW system is novel.

[10] Alternatively, CGP may increase the number of columns to the left a certain PE can access, but this would further increase the multiplexer size and design complexity.

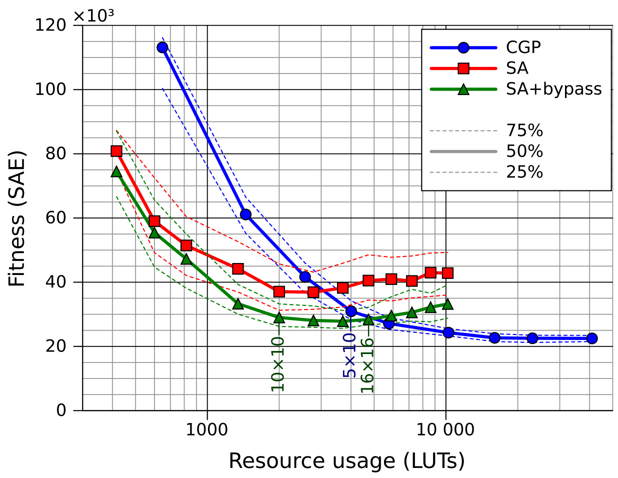

Fig. 23 Results obtained with the modified PEs (labeled as "SA+bypass"), compared with the ones obtained with CGP and the unbypassed version of SA. As it can be seen, the LUT usage for the new SA is identical to that of the old one, but the results are much better

and thus a PE is represented with 3 genes (similarly to CGP, except in this case the two extra genes represent the behavior of the outputs rather than the inputs).

The results obtained with this new PE architecture are shown in Fig. 23. As it can be seen, although this new SA topology also reaches a performance limit at a size of 10 × 10 above which it barely improves, the results are much better than before, outperforming those of CGP for sizes of 5 × 10 but requiring only 50% as many resources. Additionally, the results are much less disperse than in the former SA implementation, matching those of CGP, which suggests that the EA converges better for this new approach. Finally, the odd effect of upscaling having a negative impact on the results manifests much later.[11]

7 Conclusions

This paper compares the scalability of CGP and SA topologies for EHW in the context of performance and resource usage. In addition, it proposes a modification on the SA implementation as used in previous work which greatly improves its performance without increasing its resource usage.

If resources are scarce (as is typically the case for embedded systems), SA is a clear winner for sizes up to 10 × 10 PEs, being able to achieve better results than

[11] Further experiments have shown that larger arrays eventually outperform small ones with long enough evolution times, which suggests that the culprit for this effect is the excessively large search space for the current EA. Future work may focus on improving this EA in order to achieve better results in a shorter time.

CGP in half the resource usage. Therefore, SA would be the preferred topology for applications in which it is desirable to use a small amount of resources for the evolvable part of the system, leaving the rest for other elements of the system; or when multiple processing arrays are to be implemented in parallel [5, 26]. However, in the current use case and with the current EA, the SA topology reaches a performance limit at 10×10 PEs above which it is not able to improve.

For very large array sizes, CGP seems to scale better than SA in terms of performance, being able to improve for sizes of up to 10×20 PEs and not losing quality with larger sizes; however, CGP may also eventually show this behavior for larger sizes (which have not been explored due to limitations in the simulated model). In any case, the results obtained with very large CGP architectures are not much better than those obtained with SA.

Furthermore, there are other factors aside from resource usage that make CGP scalability less practical to implement. First, it involves changing the size of the input multiplexers. Additionally, 2D-scalable systems such as the one proposed in [9] are easy to implement using an SA since the architecture of each fragment is independent of the array size, but cannot be done with CGP since increasing its size would modify the architecture and interconnection of the PEs.

Both SA and CGP stop improving above a certain size, which may indicate either that the employed EA with the current parameters is not good enough for large array sizes or that this approach with this set of PE functions is limited. The former conclusion is backed by the fact that leaving the EA for a longer time shows an improvement in the results. Future work may focus into improving the EA to reach the best possible results faster and make a more accurate evaluation of the maximum capabilities of both architectures, as well as exploring the possibility of using a wider set of PE functions in order to improve the results.

The reduced resource usage and good performance obtained with SA motivate the future development of reconfigurable EHW systems with strict requirements, such as real-time video filtering applications, where short evolution times and a high filter throughput are desired [26]. As it was mentioned in the introduction, EHW systems constructed in this way have the property of being *self-healing*, enabling a *fault tolerant* design [32]. This can be advantageous in hostile environments such as space applications, where radiation may lead to temporary or permanent damage to the FPGA fabric.

Finally, it is worth noting that, although the use case studied in this article is limited to image filtering, this kind of system is likely to be extensible to other applications such as feature extraction, classifiers, or processing of digital signals other than images (such as audio, mechanical sensors, or EEG signals). In several of these cases, this could require using extended precision formats (such as 16 or 32 bits). In that case, given the modularity of the PE architecture used in this work, the resource usage of the resulting system would be proportionally equivalent to the values described here.

Acknowledgements This work was partially supported by the Spanish Ministry of Economy and Competitiveness under the project REBECCA (Reference TEC2014-58036-C4-2-R), and the FPI grant program of said Ministry (Grant No. BES-2012-060459). The LUT-based PEs were originally developed in collaboration with Ing. Roland Dobai, Ph.D., from Brno University of Technology.

References

1. D.B. Bartolini, M. Carminati, F. Cancare, M.D. Santambrogio, D. Sciuto, HERA project's holistic evolutionary framework. In *IEEE 27th International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 231–238. IEEE (2013). https://doi.org/10.1109/IPDPSW.2013.110
2. B. Blodget, P. James-Roxby, E. Keller, S. McMillan, P. Sundararajan, A self-reconfiguring Platform. In *Field Programmable Logic and Application, Lecture Notes in Computer Science*, vol. 2778, pp. 565–574. Springer Berlin, (2003). https://doi.org/10.1007/978-3-540-45234-8_55
3. F. Cancare, D.B. Bartolini, M. Carminati, D. Sciuto, M.D. Santambrogio, On the evolution of hardware circuits via reconfigurable architectures. ACM Trans. Reconfig. Technol. Syst. **5**(4), 22:1–22:22 (2012). https://doi.org/10.1145/2392616.2392620
4. F. Cancare, M.D. Santambrogio, D. Sciuto, A direct bitstream manipulation approach for virtex4-based evolvable systems. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 853–856. IEEE (2010). https://doi.org/10.1109/ISCAS.2010.5537429
5. R. Dobai, K. Glette, J. Torresen, L. Sekanina, Evolutionary digital circuit design with fast candidate solution establishment in field programmable gate arrays. In *2014 IEEE International Conference on Evolvable Systems (ICES)*, pp. 85–92 (2014). https://doi.org/10.1109/ICES.2014.7008726
6. R. Dobai, L. Sekanina, Image filter evolution on the Xilinx Zynq platform. In *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 164–171. IEEE (2013). https://doi.org/10.1109/AHS.2013.6604241
7. R. Dobai, L. Sekanina, Low-level flexible architecture with hybrid reconfiguration for evolvable hardware. ACM Trans. Reconfig. Technol. Syst. **8**(3), 20:1–20:24 (2015)
8. A.E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing. Natural Computing Series* (Springer, Berlin, 2003). https://doi.org/10.1007/978-3-662-05094-1
9. Á. Gallego, J. Mora, A. Otero, E. de la Torre, T. Riesgo, A scalable evolvable hardware processing array. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–7. IEEE (2013). https://doi.org/10.1109/ReConFig.2013.6732266
10. K. Glette, Design and implementation of scalable online evolvable hardware pattern recognition systems. Ph.D. thesis (2008). http://urn.nb.no/URN:NBN:no-20883
11. K. Glette, P. Kaufmann, Lookup table partial reconfiguration for an evolvable hardware classifier system. In *IEEE Congress on Evolutionary Computation (CEC)*, pp. 1706–1713. IEEE (2014). https://doi.org/10.1109/CEC.2014.6900503
12. K. Glette, J. Torresen, M. Hovin, Intermediate level FPGA reconfiguration for an online EHW pattern recognition system. In *Proceedings of NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 19–26. IEEE (2009). https://doi.org/10.1109/AHS.2009.46
13. K. Glette, J. Torresen, M. Yasunaga, An online EHW Pattern recognition system applied to sonar spectrum classification. In *Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science*, vol. 4684, pp. 1–12. Springer, Berlin (2007). https://doi.org/10.1007/978-3-540-74626-3_1
14. T.G.W. Gordon, Exploiting development to enhance the scalability of hardware evolution. Ph.D. thesis (2005). https://www.bcs.org/upload/pdf/tgordon.pdf
15. G.W. Greenwood, A.M. Tyrrell, Introduction to Evolvable Hardware. IEEE Press Series Computational Intelligence. Wiley-IEEE Press (2006).https://doi.org/10.1002/0470049715
16. T. Higuchi, Y. Liu, X. Yao, (eds.) *Evolvable Hardware. Genetics And Evolution Computer Series*. Springer (2006). https://doi.org/10.1007/0-387-31238-2
17. J.R. Koza, F.H. Bennett III, D. Andre, M.A. Keane, *Genetic Programming III* (Morgan Kaufmann Publishers Inc., San Francisco, 1999)
18. J.R. Koza, M.A. Keane, M.J. Streeter, W. Mydlowec, J.Y. Guido, *Genetic Programming IV, Genetics Programming Series*, vol. 5. Springer (2005). https://doi.org/10.1007/b137549
19. H.T. Kung, C.E. Leiserson, *Systolic Arrays for (VLSI)* (CMU-CS. Carnegie-Mellon University, Department of Computer Science, 1978)

20. S.Y. Kung, On supercomputing with systolic/wavefront array processors. Proc. IEEE **72**(7), 867–884 (1984). https://doi.org/10.1109/PROC.1984.12944
21. T. Martinek, L. Sekanina, An evolvable image filter: experimental evaluation of a complete hardware implementation in FPGA. In *Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science*, vol. 3637, pp. 76–85. Springer, Berlin (2005). https://doi.org/10.1007/11549703_8
22. J.F. Miller (ed.), *Cartesian Genetic Programming. Natural Computing Series* (Springer, Berlin 2011). https://doi.org/10.1007/978-3-642-17310-3
23. J.F. Miller, P. Thomson, Cartesian genetic programming. In *Genetic Programming, Lecturer Notes In Computer Science*, vol. 1802, pp. 121–132. Springer, Berlin (2000). https://doi.org/10.1007/978-3-540-46239-2_9
24. J. Mora, Á. Gallego, A. Otero, E. de la Torre, T. Riesgo, Noise-agnostic adaptive image filtering without training references on an evolvable hardware platform. In *2013 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 182–189 (2013). http://ieeexplore.ieee.org/document/6661538/
25. J. Mora, A. Otero, E. de la Torre, T. Riesgo, Fast and compact evolvable systolic arrays on dynamically reconfigurable FPGAs. In *10th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–7. IEEE (2015). https://doi.org/10.1109/ReCoSoC.2015.7238087
26. J. Mora, E. de la Torre, Accelerating the evolution of a systolic array-based evolvable hardware system. Microprocess. Microsyst. **56**, 144–156 (2018). https://doi.org/10.1016/j.micpro.2017.12.001
27. A. Otero, Á. Morales-Cas, J. Portilla, E. de la Torre, T. Riesgo, A modular peripheral to support self-reconfiguration in SoCs. In *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, pp. 88–95. IEEE (2010). https://doi.org/10.1109/DSD.2010.100
28. R. Salomon, H. Widiger, A. Tockhorn, Rapid evolution of time-efficient packet classifiers. In *Proceedings of IEEE Congress Evolutionary Computation (CEC)*, pp. 2793–2799. IEEE (2006). https://doi.org/10.1109/CEC.2006.1688659
29. R. Salvador, Parametric and structural self-adaptation of embedded systems using evolvable hardware. Ph.D. thesis (2015). http://oa.upm.es/39354/
30. R. Salvador, A. Otero, J. Mora, E. de la Torre, T. Riesgo, L. Sekanina, Evolvable 2D computing matrix model for intrinsic evolution in commercial FPGAs with native reconfiguration support. In *2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 184–191 (2011). https://doi.org/10.1109/AHS.2011.5963934
31. R. Salvador, A. Otero, J. Mora, E. de la Torre, T. Riesgo, L. Sekanina, Self-reconfigurable evolvable hardware system for adaptive image processing. IEEE Trans. Comput. **62**(8), 1481–1493 (2013). https://doi.org/10.1109/TC.2013.78
32. R. Salvador, A. Otero, J. Mora, E. de la Torre, L. Sekanina, T. Riesgo, Fault tolerance analysis and self-healing strategy of autonomous, evolvable hardware systems. In *2011 International Conference on Reconfigurable Computing and FPGAs*, pp. 164–169 (2011). https://doi.org/10.1109/ReConFig.2011.37
33. L. Sekanina, Virtual reconfigurable circuits for real-world applications of evolvable hardware. In *Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science*, vol. 2606, pp. 186–197. Springer, Berlin (2003). https://doi.org/10.1007/3-540-36553-2_17
34. L. Sekanina, Handbook Of Natural Computing, Chap. Evolvable Hardware, pp. 1657–1705 (Springer, Berlin, 2012). https://doi.org/10.1007/978-3-540-92910-9_50
35. A. Thompson, Silicon evolution. In *Proceedings of the 1st annual conference on Genetic Programming*, pp. 444–452. MIT Press (1996)
36. J. Torresen, G.A. Senland, K. Glette, Partial reconfiguration applied in an on-line evolvable pattern recognition system. In *NORCHIP*, pp. 61–64. IEEE (2008). https://doi.org/10.1109/NORCHP.2008.4738283
37. G. Tufte, P.C. Haddow, Evolving an adaptive digital filter. In *Proceedings of 2nd NASA/DoD Workshop on Evolvable Hardware*, pp. 143–150. IEEE (2000). https://doi.org/10.1109/EH.2000.869352
38. A. Upegui, E. Sánchez, Evolving hardware with self-reconfigurable connectivity in Xilinx FPGAs. In *Proceedings of 1st NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pp. 153–162. IEEE (2006). https://doi.org/10.1109/AHS.2006.38
39. Z. Vasicek, L. Sekanina, An evolvable hardware system in Xilinx Virtex II Pro FPGA. IJICA **1**(1), 63–73 (2007). https://doi.org/10.1504/IJICA.2007.013402

40. Z. Vasicek, L. Sekanina, Hardware accelerators for cartesian genetic programming. In *Genetic Programming, Lecture Notes in Computer Science*, vol. 4971, pp. 230–241. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-78671-9_20
41. Z. Vasicek, L. Sekanina, Hardware accelerator of cartesian genetic programming with multiple fitness units. Comput. Inf. **29**(6), 1359–1371 (2010). http://www.cai.sk/ojs/index.php/cai/article/view/149/126
42. Z. Vasicek, M. Zadnik, L. Sekanina, J. Tobola, On evolutionary synthesis of linear transforms in FPGA. In *Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science*, vol. 5216, pp. 141–152. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-85857-7_13
43. J. Wang, Q.S. Chen, C.H. Lee, Design and implementation of a virtual reconfigurable architecture for different applications of intrinsic evolvable hardware. IET Comput. Digit. Tech. **2**(5), 386–400 (2008). https://doi.org/10.1049/iet-cdt:20070124
44. J. Wang, C.H. Piao, C.H. Lee, Implementing multi-VRC cores to evolve combinational logic circuits in parallel. In *Evolvable Systems: From Biology to Hardware, Lecture Notes in Computer Science*, vol. 4684, pp. 23–34. Springer, Berlin (2007). https://doi.org/10.1007/978-3-540-74626-3_3
45. Xilinx Inc., Virtex-5 FPGA User Guide (UG190) (2012)
46. Xilinx Inc., 7 Series FPGAs Configurable Logic Block (UG474) (2014)
47. R.S. Zebulum, M.A.C. Pacheco, M.M.B.R. Vellasco, *Evolutionary Electronics: Automatic Design of Electronic Circuits and Systems by Genetic Algorithms, 1 edn. no. 22 in International Series In Computer Intelligence* (CRC Press Inc., 2001)

Affiliations

Javier Mora[1] · **Rubén Salvador**[2] · **Eduardo de la Torre**[1]

Rubén Salvador
ruben.salvador@upm.es

Eduardo de la Torre
eduardo.delatorre@upm.es

[1] Centre of Industrial Electronics (CEI), Universidad Politécnica de Madrid, C/ José Gutiérrez Abascal 2, 28006 Madrid, Spain

[2] Research Center on Software Technologies and Multimedia Systems for Sustainability (CITSEM), Universidad Politécnica de Madrid, C/ Alan Turing 3 – Edif. La Arboleda, 28031 Madrid, Spain