Accelerating the evolution of a systolic array-based evolvable hardware system

Javier Mora*, Eduardo de la Torre

Centre of Industrial Electronics, Universidad Politécnica de Madrid, Madrid, Spain

ABSTRACT

Evolvable hardware is a type of hardware that is able to adapt to different problems by going through a previous training stage which uses an evolutionary algorithm to find an optimized configuration. This configuration can be achieved through dynamic partial reconfiguration of an FPGA. Having a short time for the training stage is critical for the system to be able to adapt to changing conditions in real time. However, one of the problems of evolvable hardware based on dynamic partial reconfiguration is its long evolution time, mostly due to its slow re-configuration speed. This can make such systems unsuitable for applications which require adaptation in a few seconds. Nevertheless, different reconfiguration and evolution techniques can substantially reduce the time taken by an evolvable hardware system to evolve for a specific problem.

In this article, a system initially able to evolve in 8 minutes is optimized using multiple techniques (re-configuration methodology, evolutionary algorithm optimization, and parallelisation) so that it is able to obtain similar results in less than 2 s, achieving a speedup of near 300 times. Extensive experimental results prove the benefits of such techniques.

Keywords:
FPGA
Evolvable hardware
Dynamic partial reconfiguration
Evolutionary algorithm
Systolic array
LUT

1. Introduction

Evolvable hardware (EH) systems are configurable hardware systems which are able to adapt to different problems. Unlike classical system design, where the designer decides or calculates the structure and configuration of the system based on the problem specifications, EH uses an evolutionary algorithm (EA) to tune its parameters or structure in order to find the optimal configuration for a certain problem according to a set of training samples.

These training samples are representative examples of the problem that needs to be solved. For instance, a system whose purpose is to remove a certain type of noise from an image stream would use a noisy image as a training input and the same image without noise as a training reference, and 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 tune the hardware so that the result of processing the training input with the EH system is as similar as possible to the training reference. Once the EH has been tuned for a specific problem, it is able to process actual input for which the reference is unknown.

Obtaining a training input and reference can be done in several ways. For example, the training input could be retrieved from the actual input of the system, and then processed using a known algorithm in software in order to obtain the desired output (this process can be very slow). Once finished, the obtained image will be used as training reference together with the training input in an EA which will tune the EH to emulate the work of the software. Once the EH has been tuned, it will be able to perform a similar task to the software but with a speed which is typically much higher. This way, the EH acts as a self adaptive hardware accelerator that mimics a software task.

The training input and reference for the case of noise removal can also be obtained by getting a generic noise free image as the training reference, and adding a specific type of noise to it in order to obtain the training input. However, this task needs knowledge of the type of noise and images that will need to be filtered. Nevertheless, previous work [1] shows that this can also be done by using the system input directly, and relying on the random nature of noise and the non locality of the filter to use two noisy inputs as training input and reference (Fig. 1).

The evolution can be extrinsic or intrinsic, depending on whether the candidate solutions are evaluated on a simulated model or on the EH system itself. The advantage of intrinsic evolution is that it makes the EH 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.

Intrinsic evolution also removes the need to use a software
simulation of the hardware in order to perform the evolution, so only the EA itself needs to be run in software. Nevertheless, the EA can be simple enough to run in an embedded processor next to the EH, or even implemented purely in a specialized hardware module; therefore the system can be implemented in a SoC, making it completely autonomous.

FPGAs are a very good platform for implementing EH, specially those equipped 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 EH system as a piecewise circuit composed by multiple processing elements (PEs), each of which implements a simple task, and whose functionality can be individually changed by replacing it with a different PE using DPR.

A problem of DPR is that it is typically very slow, and is thus normally used only for coarse grain applications that are seldom reconfigured, as its intensive use in an EA would make the training times excessively long. Nevertheless, implementations that accelerate the reconfiguration by using hardware accelerated reconfiguration engines or reducing the amount of logic to be reconfigured exist [2,3], leading to relatively fast reconfiguration times that take only a small fraction of the total time used by the training stage.

During the training stage, the system is either inoperative or working at suboptimal performance; therefore, it is desirable that the time taken by this stage is as short as possible. Therefore, making the training stage of DPR based EH shorter can greatly increase the amount of applications where such a system can be applied. This can be achieved through the combination of three approaches:

- Reducing the number of candidate solutions that need to be generated and evaluated by the EA until an adequate solution is found.
- Reducing the reconfiguration time needed to implement a certain solution.
- Reducing the evaluation time needed to test a certain solution once it has been configured.

The time overhead taken by the software leading the EA is typically small compared to the reconfiguration and evaluation times, specially for simple EAs.

This article describes and analyzes multiple improvements and optimization techniques that have been applied to a pre-existing EH implementation [4] in order to substantially reduce the total time needed to evolve for a particular problem, some of which were already used in [5]. By applying these techniques, the evolution time has been reduced from 8 minutes to less than 2 seconds.

The rest of the article is organized as follows: Section 2 introduces the state of the art and possible alternatives. Section 3 describes the initial implementation of the system to be optimized. Section 4 describes the hardware improvements made on both the hardware architecture and the reconfiguration engine with the aim of reducing both reconfiguration and evaluation times, as well as reducing the resource usage of the architecture. In Section 5, certain modifications to the original EA are made in order to improve its efficiency, reducing the number of candidate solutions generated and evaluated in order to obtain good results. Section 6 takes advantage of the reduction in resource usage performed in Section 4 to parallelize the EA across multiple evaluation units, which further decreases the evaluation time. Finally, Section 7 shows the conclusions of the article and summarizes the improvements achieved in each section.

This article is structured in an incremental approach, where each improvement is analyzed before moving on to the next one, since most of the times an improvement is justified by the results of the previous one. There is no separate section for the results; instead, these are shown at the end of each subsection.

2. Technical background and previous work

Common EH based processing systems consist of a large number of basic processing units, known as processing elements (PEs), which are interconnected in a specific manner. Each of these PEs has a certain number of inputs coming from the system input or from other PEs, implements a specific operation on the data it receives from these inputs, and sends the processed result to other PEs, typically registering the result in order to create a pipelined data processing architecture. The mission of the EA is to determine which operation will be performed by each PE and how the PEs will be interconnected; these parameters constitute a specific candidate solution.

This section describes different topologies frequently used in EH, as well as multiple common techniques for changing its configuration.

2.1. Interconnection topologies

Given that allowing every PE to get its inputs from any other possible PE in the system would lead to excessively complex routing (which is generally bad for FPGA design) and to having excessively big multiplexers at the inputs of the PEs, the way in which PEs can interconnect is usually restricted so that only a few possible interconnections are allowed.

One of these interconnection topologies is the Cartesian genetic programming (CGP) [6], which consists of a series of PEs arranged in columns, as seen in Fig. 2. Each of these PEs can take data from the primary input and the columns to the left, and usually implements a stateless simple function (typically 1 bit logic gates). In order to further simplify the hardware implementation in terms of multiplexers and routing, as well as the search space for the EA, the number of inputs available to a certain PE can be constrained to a maximum number of columns to the left (typically one column, to avoid large multiplexers).

Although PEs in CGP typically implement 1 bit logic gates as their processing function [6,7], authors have shown that this is inefficient for evolvable image filters, and replace these basic logic functions with more complex functions such as 8 bit adders [8], which do not need to yield an exact result but can be simplified approximations [3]. Other authors go one step further and create complex PEs on the fly by combining primitive functions, a technique known as embedded CGP (ECGP) [9].
Fig. 3. Example of a $3 \times 3$ systolic array with 3 inputs and 1 output. Notice how the routing has been simplified and the multiplexers removed except for the ones at the input and output of the system.

The main problems with this topology are 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. Additionally, the complexity of this topology varies with scalability, as larger multiplexers are needed for bigger array sizes.

Multiplexers also have the drawback of using a high amount of re sources. 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 [3] (9 inputs + 4 PEs) requires 4 LUTs per output bit [10], 64 LUTs in total for 2 input multiplexers. Therefore, multiplexers alone would represent 89% of the resource usage for this topology.

Another topology which does not suffer this problem is the systolic array, first defined in [11] as a generic computing engine, and used as a reconfigurable fabric for implementing EH in [12]. It was originally intended for complex PE operations but it can be used with simpler PEs as well. Opposite to CGF, inputs to each PE are fixed, connecting each of them to its neighbors (Fig. 3).

This topology simplifies the routing between processing elements, allowing for shorter data paths and thus lower delay. Also, smaller logic resources per every PE allows PEs to be closer between neighbors. Additionally, it removes the need of having a multiplexer at the input of every PE (except at the system input), thus again reducing the delay as well as the resource usage. Its simplicity is also an advantage in dynamically scalable systems such as [4].

The disadvantage of this solution is the degradation of 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.

2.2. Reconfiguration methods

As said before, PEs must be able to switch between different functions according to the configuration of a specific solution.

The most straightforward way to achieve this is by simply implementing all possible circuits and using a multiplexer to select which function the PE uses (Fig. 4), in a similar way to how an ALU works. This is known as virtual reconfigurable circuit (VRC) [13], and has the advantage of being highly portable (independent of the FPGA used). However, this approach is considerably resource and energy consuming, since all the possible functions have to be implemented at once, and the extra multiplexer to select the used function has the same problems as CGF has: it introduces extra delay and resource usage.

This approach is used in [14] in combination with CGF in order to implement an evolvable image filter.

An alternative to VRCs consists in using DPR if the FPGA supports it. With this methodology, rather than having to implement all possible PE circuits physically in the FPGA, only one of them is present at a time. A reconfiguration engine will later be able to replace this circuit with another one stored in a PE library by partially reconfiguring the area of the FPGA corresponding to that PE (Fig. 5).

This approach reduces considerably the amount of resources used by a single PE and potentially improves its maximum frequency of operation, at the expense of making the system specific to a certain FPGA model. Additionally, DPR introduces a reconfiguration time overhead (specially when a software driven reconfiguration engine is used, in which case the reconfiguration can take several milliseconds [15]), unlike VRCs where this time is negligible since it would only involve changing the input of some multiplexers. Nevertheless, in [2] the reconfiguration time is still small compared to the evaluation time, thanks to the usage of efficient hardware based reconfiguration engines.

The design of modules interchangeable by means of partial reconfiguration requires the input and output nets of the module to be compatible with the ones on the static system. Xilinx tools provide a solution for this requirement in their partial reconfiguration flow [16]. However, this flow does not support using the same circuit on different positions (relocation) even though this is theoretically possible due to the uniformity of FPGAs.

An alternative to Xilinx's partial reconfiguration flow that allows relocation is the use of bus macros (as is done in [12]), which route nets in a replicable way, at the cost of extra delay and resource usage. More recently, the usage of bus macros has been replaced by custom tools such as Dreams [17] or GoAhead [18] which do not present this resource overhead. The Dreams tool is used in [4] to implement a scalable systolic array of up to 8 × 7 PEs, each with a size of 5 × 1 CLBs.

A third way to change the functionality of a circuit is to change the content of the FPGA LUTs that are used to implement logic functions, which is not as flexible as changing the complete circuit including inter connection nets, but can still be a good solution if the PEs are similar, and is easier to accomplish given that no special routing considerations have to be taken. This was done in [19] on a Xilinx Virtex II Pro by temporarily using the LUTs as shift registers in order to write a new content. However, the amount of LUTs that can work as shift registers has decreased in more modern Xilinx FPGA families, and currently only 25 30% of the LUTs in Virtex 5 [20] and 7 series [10] FPGAs have this functionality, unlike Virtex II Pro which allowed this in all of its LUTs [21].

[3] uses a similar LUT reconfiguration approach in a CGP topology, but using DPR instead of relying on a shift register functionality, thus being able to use all the LUTs in the FPGA, not only the aforementioned 25 30% of them. This implementation achieves a remarkable speed of 8700 candidate evaluations per second. This same approach is combined with parallelization of multiple CGP arrays in [22] in order to obtain speeds of 30 000 candidate evaluations per second, completing the evolution in only 10 s. In [5], the same approach is applied to an SA architecture, using it to reconfigure input and output multiplexers as well.
3. Initial system

The system this work targets is an evolvable image filter based on a reconfigurable systolic array (SA), which uses dynamic partial reconfiguration (DPR) in order to change its functionality. The reconfiguration is driven by an evolutionary algorithm (EA) which runs as a software program on a MicroBlaze processor. The whole system is implemented on a Xilinx Virtex 5 LX110T 1 FPGA.

3.1. Hardware

The image filter is based on a systolic array (Fig. 6) which receives data from a 128 × 128 pixel grayscale image by using a 3 × 3 pixel sliding window which moves one pixel per clock cycle to the right. The SA operates on the data of this window, generating one pixel value per clock cycle at its output. This output stream of pixels is then stored as the filtered image.

Additionally, the output is compared pixel by pixel with a reference image, using the sum of absolute errors (SAE) as a metric to measure the quality of the filter:

\[
SAE = \sum_{i=0}^{127} \sum_{j=0}^{127} |output_{ij} - reference_{ij}|
\]

This metric is used as the fitness criterion on an evolutionary algorithm (EA) which is used to train the filter for a specific problem. This EA runs on a MicroBlaze processor.

The SA is structured as an array of processing elements (PEs) as depicted in Fig. 6. 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, subtraction, constant value, etc. The list of available functions is shown in Table 1. Every PE on the north and west borders of the SA takes its input from one of the 9 pixels of the input window, and the SA output is obtained from one of the PEs on the east border. The function implemented by each of the PEs, the pixel selected at each of the inputs, and the output selected as the result constitute the configuration of the array.

This particular system was first described in [2], and relied on dynamic partial reconfiguration (DPR) of a Xilinx Virtex 5 FPGA in order to change the configuration of the processing elements, and multiplexers for the inputs and output. DPR was done using a custom hardware accelerated reconfiguration engine able to perform DMA to an external DDR2 SDRAM where the partial bitstreams containing the FPGA configuration data are stored, rather than relying on the IP core provided by Xilinx (XPS HWICAP) which is fully software driven and lacks DMA capabilities. Additionally, although the nominal frequency of the FPGA’s internal configuration access port (ICAP) is 100 MHz, this engine was overclocked in order to reach frequencies of up to 250 MHz, since it was shown in [23] that overclocking the ICAP is possible.

The SA was built with PEs of a size of 20 configurable logic blocks (CLBs) in height and 2 CLBs in width (40 CLBs in total). This size was motivated by the size of a reconfiguration frame, which spans 20 CLBs vertically, not allowing the direct reconfiguration of regions smaller than this size. Additionally, routing restrictions made it difficult to implement these PEs in a size narrower than 2 CLBs. All this made it difficult to implement SAs larger than 4 × 4 PEs.

In order to overcome this size limitation, a new implementation was developed which had a PE size of only 5 × 1 CLBs [4], allowing SAs as large as 8 × 7 PEs. This was made possible by using a custom routing tool [17] and a feature of the reconfiguration engine which allows reading the configuration of a region with a height of one frame and write it back modifying only a fragment. However, this approach introduced some drawbacks:

- It doubles the reconfiguration time, since it requires reading in addition to writing.
- The readback operation presented timing errors when overclocking the ICAP, so the frequency had to be reduced to 100 MHz.
- Even though the PE is now only 5 CLBs big, a whole column of 20 CLBs needs to be read and written in order to reconfigure a fragment of it.

Therefore, although the new PEs are smaller than the old ones, this improvement in size caused the reconfiguration time to increase.

This SA was able to operate at frequencies of 250 MHz. However, due to limitations of the logic that interfaces with it, the whole system frequency was set to 100 MHz to prevent timing errors.

With this implementation, the system was able to generate and evaluate approximately 2000 candidate solutions per second.

3.2. Evolutionary algorithm

The algorithm employed 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 3 × 3 sliding 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 algorithm used is a (1 + 8) EA (Algorithm 1), which is described below.

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
parameter $generations := 120000$
parameter $\lambda := 8$

procedure EVOLUTION
  initialize parent
  evaluate parent
  for $g := 1 \ldots generations$ do
    evolveGeneration parent
  end for
  return parent
end procedure

procedure evolveGeneration parent
  for $i := 1 \ldots \lambda$ do
    children$[i] \leftarrow$ parent
    mutate children$[i]$
    evaluate children$[i]$
  end for
  bestChild $\leftarrow$ children[lowest fitness]
  if fitness(bestChild) $\leq$ fitness(parent) then
    parent $\leftarrow$ bestChild
  end if
end procedure

Algorithm 1. $\lambda$-EA with $\lambda = 8$.

PE, in order to obtain an identity filter. This candidate solution is evaluated by filtering a training image and comparing the filter output (currently identical to its input) with a reference image, obtaining the solution's fitness as the SAE between the obtained output and the reference.

After initialization, an iterative process takes place. On each iteration (generation), $\lambda$ copies (children) are made from the current configuration (parent). Each of the copies is then mutated with a mutation rate of $K = 3$. This means that 3 random genes are selected and replaced by a random value. Then, the $\lambda$ mutated solutions are evaluated by re-configuring the SA and filtering the training image, obtaining their fitness. Finally, the best solution among the parent and children (the one with the lowest fitness) is selected as the new parent for the next generation. In case of tie, the children have preference over the parent.

The iterative process is repeated for 120 000 generations, evaluating a total of 960 000 candidate solutions.

For this study, the EA has been used to obtain a filter for removing salt and pepper noise from an image. This was achieved by using a Lena test image (Fig. 7) to which a 20% of salt and pepper noise has been artificially added as the training image, and the original image without noise as the reference. Once the EA has concluded, the obtained filter is able to remove this type of noise from other images [24]. Previous works [1,24] show that this kind of system evolves correctly in dependently of the training set for several types of noise, so no other images nor noise types have been tested in this work.

In order to obtain statistically meaningful results, the evolution process was repeated 1000 times with different initial random seeds. The fitness value is registered every 1500 generations, and different percentiles are then calculated for each point in the evolution. The result of this is shown in Fig. 8. As can be seen, the median of the results at the end of the evolution is approximately 40000. As a reference, the original image has an SAE with respect to the reference of 416297, and the image filtered with a median filter using a $3 \times 3$ pixel window has an

---

*Salt and pepper noise consists in replacing randomly selected pixels (in this case, 20% of them) with black or white pixels.
The current EA shows a quick progress at the beginning, but this progress is slowed down at the end. This may be because this kind of EA tends to get stuck at local optima, and once this happens it will typically take a long time to find a better solution. Therefore, finding strategies to avoid local optima or minimize their effect may improve the efficiency of the EA.

In the next sections, several modifications are applied to this system in order to reduce the evolution time.

4. Hardware improvements

4.1. Reconfiguration engine improvement

In order to overcome the drawbacks of the reconfiguration engine described in Section 3.1, a new engine targeting fine grain reconfiguration has been designed. The main difference with the former one is that, rather than relying on ICAP readback for the modification of a fragment of a whole CLB column, the new engine simply rewrites all the PEs of the column, leaving to the software the task of memorizing which PEs were on that column before. Therefore, the software will only have to pass a list of memory addresses where the partial bit streams of the respective PEs are stored.

In order to access the partial bitstream data quickly, the external DDR2 SDRAM has been replaced with an internal block RAM (BRAM), which can be accessed in arbitrary order at one word per clock cycle. This type of memory is much more limited in size than the external DDR2 SDRAM; however it is enough for fine grain reconfiguration applications, given the small size of the PEs, which can be stored in as little as 128 KB.

The reconfiguration engine has been simplified and optimized, and operates at a speed of 200 MHz. This, together with the removal of the readback stage, have made the reconfiguration time 4 times shorter.

4.2. PE optimization: LUT reconfiguration

Another optimization of the hardware system has involved redesigning the PEs so that they all have the same structure, differing only on the logic configuration. This removes the need of reconfiguring the interconnections, needing only to reconfigure the lookup tables (LUTs) that implement the logic functions of the circuit, which amounts to only 8 of the 36 configuration frames needed to reconfigure a whole CLB column in the Virtex 5 family, thus making the reconfiguration time considerably shorter. Additionally, it removes the need of special routing considerations, which allows implementing the whole PE in a size as small as 2 × 1 CLBs, needing only 128 CLBs to implement an SA of 8 × 8 PEs.

This new version of the PEs was developed in collaboration with Prof. Roland Dobai from the Brno University of Technology. The structure of these PEs is shown in Fig. 10, and is implemented in 16 LUTs (2 per each output bit), 8 FFs and two 4 bit carry chains, all of which fits on 4 Virtex 5 slices. By providing the right LUT configuration, this PE structure can implement all of the functions in Table 1. In order to further reduce the amount of logic to reconfigure, unused LUT inputs are tied to zero; this reduces the number of frames to reconfigure to 3 (1 for the first LUT and 2 for the second one) rather than all 8 of them.

The input selectors on the north and west borders of the SA are tied to zero; this reduces the number of frames to reconfigure to 3 (1 for the first LUT and 2 for the second one) rather than all 8 of them.

The input selectors on the north and west borders of the SA are implemented in a similar way (Fig. 11) and changed using DPR as well, rather than using actual multiplexers which would require additional logic for changing their control registers. In order to simplify the system, only 3 of the 9 pixels from the input window are fed to each input selector; the remaining 6 are obtained by delaying this input by a varying amount of clock cycles. Input selectors are pipelined in order to be synchronous with the pipelining of the PEs.

The output selector is built by cascading LUTs as shown in Fig. 12, and is pipelined as well. This way of pipelining the PEs, inputs, and output is done in order to synchronize the circuit so that data from
input windows at different clock cycles does not mix (Fig. 13) and its behavior is functionally equivalent to a purely combinatorial one.

The reduced size of the new PEs (only 2 CLBs) allows for a very compact implementation, resulting in short interconnections that provide a good timing behavior. This makes the new SA implementation able to operate at 400 MHz, which is close to the 600 MHz theoretical maximum clock frequency achievable on this FPGA [25].

However, some parts of the logic interfacing with it such as the comparator (SA8 computing unit) cannot reach this speed. In order to overcome this limitation, the system has been divided into multiple clock domains, using dual clock asymmetric FIFOs to synchronize the data, as shown in Fig. 14. This allows having the SA work at 400 MHz while the BRAMs and the comparator work at 100 MHz, processing 4 pixels in parallel in order to keep the throughput at 400 megapixels per second.

Additionally, the reduced size of the PEs and the fact that only a few frames need to be reconfigured reduces the required size of the BRAM to less than 2 KB, making it able to fit in a single RAMB36 block (which has a capacity of 4 KB).

4.3. Evaluation of results and conclusions

By redesigning the PE structure so that all its configuration relies on LUT content and implementing an improved and optimized DPR engine that targets fine grain DPR, the reconfiguration times have been reduced considerably. Additionally, the design has become much more compact, which makes it resource efficient and able to operate at high frequencies.

With these improvements, the new implementation has increased the evaluation speed from 2000 to 19 000 evaluations per second, making the evolution time almost 10 times shorter. This hardware is able to perform the same evolution process in 50 s.

The reduction of the PE size and the removal of routing restrictions via the use of LUT reconfiguration allow implementing SAs of a larger size (tested to sizes up to 24 × 24) or putting multiple arrays working in parallel.

The advantage of using DPR to reconfigure LUTs over other techniques such as using LUTRAM or SRL is that it allows maximizing the local occupancy, i.e., using 100% of the LUTs on the FPGA region used by the SA, rather than being restricted to a small number of them.

5. EA improvements

Another way to tackle the problem of having a long evolution time is to optimize the EA so that it reaches the solution faster, needing fewer evaluations in order to obtain the same result. In this section, different evolution strategies are studied and compared.

5.1. Mutation rate

Previous work made in this research line has typically used mutation rates of $K=3$. This mutation rate was chosen experimentally by performing a few tests on a $4 \times 4$ SA, and it was observed that $K=3$ yielded good enough results. However, as the array size increases, it becomes important to analyze the mutation rate to see if improvements can be obtained by varying it. Additionally, it should be noted that larger mutation rates involve performing a larger number of reconfigurations, increasing the reconfiguration time and thus slowing down the evolution. Therefore, it would be interesting to make the mutation rate as small as possible without affecting the obtained quality.

Fig. 15 shows the results of the EA with different mutation rates, performing 1000 runs for each of them. Since the EA is still rather slow and a high number of tests are involved, the EA has been run for only 30 000 generations (240 000 evaluations).

As can be seen, the best results are obtained for mutation rates of 2
Fig. 14. Clock domain division of the system, differentiating the clock domains of the MicroBlaze processor (1), the BRAM reading and writing (2), the systolic array itself (3), and the comparator (4), which is able to process multiple pixels in parallel.

Fig. 15. Results obtained for different mutation rates, showing the median value of the fitness (top) and the fitness distribution after evolving for a total number of 240 000 evaluations (bottom).

and 3, with nearly identical results. A mutation rate of 1 gives very poor results, causing the evolution to rapidly stall. Mutation rates of 4 or more make the evolution converge more slowly, and thus yield worse results for a same evolution time. Additionally, as it has been stated, high mutation rates cause a longer reconfiguration time. Therefore, given that values of 2 and 3 produce the best results, a mutation rate of 2 becomes a more reasonable choice.

In order to ensure that the difference between these results is statistically meaningful and not a result of an excessively small sample size (even for a size of 1000 values), a two-tailed Student's t test has been performed. This test yielded a p value of 0.92 for K=2 versus K=3, not indicating statistical significance, and less than $10^{-36}$ for K=2 versus any other value, indicating that the apparent difference seen in Fig. 15 is statistically significant.

This applies to SAs with 8 x 8 PEs; larger sizes would probably require greater mutation rates, so separate experiments should be made for said sizes.

5.2. Offspring size

The described (1 + 8) EA is a particular case of the more general (1 + λ) EA, where λ is the offspring size, this is, the number of children tested in each generation. This is typically done in evolution strategies, which are EAs based on real numbers where a small change typically causes an improvement or regression with similar probabilities. However this is not the case for the EA used here, which is closer to a genetic algorithm.

Fig. 16 compares the (1 + 8) EA with a (1 + 1) EA, for the same number of evaluations (not generations), both with K = 2. As can be seen, both algorithms yield very similar results. This suggests that the number of children does not affect the outcome of the EA for the same number of evaluations.

Given this result, the (1 + 1) EA has been chosen over the (1 + 8) EA in order to simplify its implementation and further modifications. Nevertheless, both strategies are equally valid, so an (1 + 8) EA (or any...
other \((1+1)\) EA could be used if needed (for example to exploit parallelizability over multiple SAs, as is done in [22]).

5.3. Increasing the diversity

The main problem of the original EA is that it is prone to getting stuck at a local optimum. This causes that the EA barely improves after a certain number of generations.

Fig. 8 showed that the dispersion of the fitness results at the end of the evolution is rather high, with their median value being at 40 000 but 90\% of the results ranging from 25 000 and 75 000. The curves near the end have a very reduced slope, therefore the evolution could be made shorter without this having a great impact on the result.

An easy way to obtain better results is to repeat the evolution multiple times, and pick the best result. For example, although the probability of getting a result below 32 000 (25\% percentile) is only 25\%, the probability of having any of 5 results below that value is 76\%. This involves running the EA multiple times, but since the gain towards the end of the evolution is small, it would be possible to shorten each of the evolutionary runs.

Fig. 17 shows the median fitness of the best of \(n\) \((1+1)\) EA evolutions for a certain number of evaluations. This value has been estimated from the data obtained for a single \((1+1)\) EA using the formula 

\[
    f_{p}(p) = f_{b}(1 - \sqrt{1 - p})
\]

with \(p = 0.5\), where \(f_{b}\) is the quantile function of the fitness for a single \((1+1)\) EA (the value for which there is a probability \(p\) of obtaining a fitness lower than that value), and \(f_{p}\) is the quantile function for the "best of \(n\)" approach.

As can be seen in Fig. 17, performing 2 5 proportionally shorter evolutionary runs yields better results than performing a single one. A higher number of evolutionary runs shows a worse performance for a small number of evaluations since excessively short individual evolutions tend to yield bad results; however, this changes as more evaluations are made. Specifically, for 960 000 evaluations, the optimum is in 10 20 evolutions.

Fig. 18 shows the experimental results of selecting the best of 12 \((1+1)\) EA evolutionary runs of 80 000 generations each. The number 12 was chosen because it is between 10 and 20, which show the best results in Fig. 17, and for reasons that will be justified in Section 6. As can be seen, the new strategy outperforms the old one after 150 000 evaluations, and can obtain results similar to the old one in half as many evaluations; additionally, the dispersion of these results is smaller.

In addition to obtaining better results, the new strategy is easily parallelizable, being able to distribute the evolutionary runs across multiple SAs which can even be on different devices; it can also be executed on a single device with a single SA by sequencing all the evolutionary runs, memorizing the final result of each one.

5.4. Promoting good evolutions

Although the strategy described in Section 5.3 reduces the effect of the EA getting stuck at local optima, there is still a risk that some of the 12 evolutionary runs get stuck, becoming of no use for the rest of the evolution. Being able to remove these evolutionary runs and giving advantage to the rest would improve the result.

A way to achieve this is to execute all the evolutionary runs con currently, performing an exchange every 1000 generations. In each exchange, the evolution with the worst fitness is considered to be likely stuck and is terminated, and the one with the best fitness is considered as a promising evolution and is forced in order to improve the chances of success (Algorithm 2).

As can be seen in Fig. 19, this modification further improves the results. The new strategy is able to achieve the results of the original one in less than 240 000 evaluations. Therefore, the evolution can be made 4 times shorter while still obtaining similar results.

5.5. Restricting mutations to a single column

Finally, a way to reduce the time spent in reconfiguration is to force all the mutations to be on the same column of the SA. Since the reconfiguration engine described in Section 4.1 writes the PE configuration in whole columns, forcing the mutations to be in the same column reduces the number of columns to reconfigure, thus decreasing the reconfiguration time by approximately a 30\%.

As can be seen in Fig. 20, the effect of this simplification on the fitness is negligible.

5.6. Evaluation of results and conclusions

By optimizing the EA, the number of evaluations (and therefore the overall evolution time) can be reduced 4 times while still achieving similar results. This reduces the total evolution time to less than 12 s.

By reducing the mutation rate and restricting all mutations to a single column, the reconfiguration time has been reduced. Currently, the reconfiguration time is small compared to the time spent filtering
parameter exchanges := 80
parameter generations := 1000
parameter runs := 12
parameter K := 1

procedure EVOLUTION
    for r := 1 ... runs do
        initialize parents[r]
        evaluate parents[r]
    end for
    for e := 1 ... exchanges do
        for r := 1 ... runs do
            for g := 1 ... generations do
                evolveGeneration parents[r]
            end for
        end for
        parents[highest fitness] := parents[lowest fitness]
    end for
    return parents[lowest fitness]
end procedure

$$80 \times 1000 \times 12 \times 1 = 960,000 \text{ evaluations}$$

- this loop can be parallelized
- see Algorithm 1

Algorithm 2. $12 \times (1+1)$-EA with exchanges.

---

![Graph 1: Fitness vs. Candidate solutions evaluated](image1)

**Fig. 19.** Comparison of the original $(1+8)$-EA, the best of $12 \times (1+1)$-EA, and $12 \times (1+1)$-EA with an exchange every 1000 generations (12,000 evaluations), all with $K=2$. The horizontal dashed line marks the median fitness obtained with the original $(1+8)$-EA with $K=2$ (42,346).

![Graph 2: Fitness vs. Candidate solutions evaluated](image2)

**Fig. 20.** $12 \times (1+1)$-EA with exchanges, with $K=2$, comparing the results of mutating Pis and selectors in any column and restricting them to a single column.

---

The image, as will be seen in Section 6, so these savings are not too beneficial for now; nevertheless they will have a greater weight once the improvements in said section have been applied.

### 6. Parallelization

After the optimizations described in Section 4, the time spent on the different stages required to obtain a single candidate solution are shown in Table 2. "Mutation" refers to the time spent by the software generating the specification of a new configuration; "reconfiguration" is the time used by the reconfiguration engine writing said specification on the FPGA fabric; "filtering" is the time spent filtering the training image with the new configuration, and "selection" is the software time used to decide if the newly obtained configuration is to be kept or if it has to be discarded.

As can be seen, the majority of the time is used by the filtering stage, so reducing this time would greatly improve the overall evolution time.

As a side note, the time spent by the filtering stage is very close to the theoretical time required by the hardware (41.3 $\mu$s). However, the

---

<table>
<thead>
<tr>
<th>Stage</th>
<th>Time ($\mu$s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutation</td>
<td>1.5</td>
</tr>
<tr>
<td>Reconfiguration</td>
<td>5.2</td>
</tr>
<tr>
<td>Filtering</td>
<td>42.0</td>
</tr>
<tr>
<td>Selection</td>
<td>0.3</td>
</tr>
<tr>
<td>Total</td>
<td>49.0</td>
</tr>
</tbody>
</table>

**Table 2**
Evolution time breakdown in microseconds for a single evaluation.
theoretical time for reconfiguring has been estimated to be less than
2.8 μs, near half the time observed in practice. This is due to the
overhead introduced by the software when driving the reconfiguration
engine.

6.1. Multiple arrays in parallel

The systolic array already operates at a frequency close to the
switching limit of the FPGA, so it is unlikely that further optimizations
would bring a great improvement in the evolution times. However, the
reduction in the resource usage of the SA achieved by using smaller PEs,
together with the removal of the routing restrictions, allows im-
plementing multiple SAs which may work in parallel, thus accelerating
the filtering stage.

However, this acceleration only benefits the time spent filtering the
image, not the mutation, reconfiguration, and selection times, which
are not parallelized since they run in software and only one ICAP can be
used at a time. Therefore, the number of candidate solutions evaluated
per second does not grow proportionally with the number of SAs, as can
be seen in Fig. 21; it grows asymptotically to a limit equivalent to
having a filtering time of 0 but keeping the other three times.

This technique is used in [5] with 8 SAs working in parallel, where
speeds of 8,000 evaluations per second were achieved. Using more
arrays would involve a great increase in the resource usage for a small
performance gain: having twice as many arrays would only improve the
speed by a 27%.

6.2. Scheduling

The previous subsection does not consider the fact that all the SAs
are idle while a single one of them is being reconfigured, and that the
microprocessor and reconfiguration engine are idle while the SAs are
filtering. However, the current implementation does not allow
launching an image filtering before the previous one has finished, since
the access to the input image is done through a single memory port.

Nevertheless, it would be possible to distribute the SAs into two
groups, as seen in Fig. 22, so that while one of the groups is filtering the
image, the other one is being mutated and reconfigured. With enough
SAs, this is equivalent to eliminating the filtering time, since it will run
in the background while the other stages are completed.

The effect of this is illustrated in Fig. 23, which shows that, although
this strategy involves underutilizing the SAs, the speed gain is linear
rather than asymptotic, overtaking the previous approach for 6 or more
SAs. The maximum speed is reached at 12 SAs. Above that point,
mutation and reconfiguration times become the limiting factors, so no
gain is obtained at all with more SAs.

The removal of the routing restrictions in Section 4 simplifies the
placement of SAs on arbitrary positions of the FPGA, which combined
with the reduction of the PE size has allowed implementing a system with 12 arrays in parallel (Fig. 24). This system is able to reach speeds of 139 000 evaluations per second, a 71% more than the 81 000 described in [5]. Notice that without the half and half scheduling described here, a system with 12 SAs would only be able to perform 95 000 evaluations per second.

Notice that this parallelization approach is not compatible with all algorithms such as a single (1 + 12) EA, which requires completing the 12 evaluations in a generation before starting the next generation; therefore, the first half of a generation would not be able to start until the second half of the previous one has concluded. This is not a problem for the current 12 × (1 + 1) EA, since the 12 concurrent evolutionary runs can be carried out independently, only having to wait to each other every 1000 generations.

6.3. Evaluation of results and conclusions

Although parallelization does not provide a linear increase in the evaluation speed, a great speedup can be obtained by implementing up to 12 arrays in parallel and using half of them while the other half is reconfigured. This accelerates the evolution 7 times with respect to having a single SA. This has led to an evolution speed of 139 000 evaluations per second, the fastest achieved so far, to the best of our knowledge.

For more than 12 SAs, mutation and reconfiguration times become the limiting factors, so the current approach cannot be further optimized unless these times are reduced. Both mutation and reconfiguration are limited by the microprocessor, so using a hardware implementation of the EA that communicates directly with the reconfiguration engine would reduce this time. If this time were reduced to the theoretical minimum required by the reconfiguration engine (2.8 μs), speeds of up to 350 000 evaluations per second could be theoretically achieved by using 30 SAs in parallel; however, these many SAs would occupy approximately 70% of the resources of the FPGA currently in use (Xilinx Virtex-5 LX110T), so it is possible that such an implementation requires a larger FPGA.

The implementation of 12 arrays in parallel has motivated the specific choice of 12 concurrent evolutionary runs in Section 5; this way, each one can be carried out independently on one SA, simplifying the implementation. Nevertheless, different numbers of concurrent evolutionary runs could be implemented, such as 24 (by performing 12 first and 12 later) or 6 (by using a (1+2) EA in order to occupy the 12 SAs on each generation). This would allow further tuning of the EA to obtain maximum efficiency, but this is out of the scope of this article.
Conclusions

In this article, multiple optimizations applied to an evolvable hardware system in order to accelerate its evolution have been described. Combining the improvements in hardware, EA, and parallelization techniques, a total speedup of the evolution of 280 times has been achieved, going from 480 seconds to only 1.73 s and achieving speeds of 139 000 evaluations per second.

As can be seen, the biggest improvement in terms of time comes from the improvement of the reconfiguration engine and the SA optimization. The optimization of the EA also provides an important speedup. These changes are independent, since the former signifies a reduction in the time used in each evaluation whereas the latter reduces the overall number of evaluations. The parallelization improvement also adds an important boost to the system performance, although in this case it is dependent on the other two improvements: it comes at the cost of a higher resource usage, which has been possible thanks to the reduction in SA resource usage; and relies on the reduction of reconfiguration times and EA parallelizability.

This article demonstrates that, although achieving a high evaluation speed is important in order to reduce evolution times, an efficient EA can also mean a significant speedup, with the additional advantage that the latter does not involve performing changes in the hardware. Therefore it would be worth considering to analyze the current EA for a finer parameter tuning (number of concurrent evolutionary runs, interval between exchanges, variable mutation rates...) or to implement other evolution techniques such as crossover.

It should be noted that, although this work has been done on a Virtex 5 FPGA, most of its results are relevant to other platforms: LUT-based reconfiguration and array parallelization can be performed on newer FPGAs such as the 7 series (and are already done by some authors [22]), and the EA improvements are independent of the platform.

Acknowledgments

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 the aforesaid Ministry [grant number BES 2012 060459]. The LUT-based version of the PEs was developed in collaboration with Prof. Roland Dobai from the Brno University of Technology.

References