Microprocessors and Microsystems 56 (2018) 144–156

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

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

ARTICLE INFO

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

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 reconfiguration 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 (reconfiguration methodology, evolutionary algorithm optimization, and parallelization) 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.

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

* Corresponding author.
E-mail addresses: javier.morad@upm.es (J. Mora), eduardo.delatorre@upm.es (E. de la Torre).

https://doi.org/10.1016/j.micpro.2017.12.001
Received 24 June 2016; Received in revised form 7 November 2017; Accepted 5 December 2017
Available online 06 December 2017
0141-9331/

Fig. 1. Evolution without a noise-free reference. Left: training input. Center: training reference. Right: evolution result. Both input and reference have a 50% of salt and pepper noise.

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].

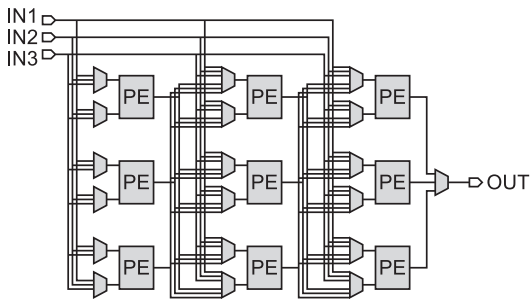

Fig. 2. Example of a 3 × 3 CGP topology with 3 primary inputs and 1 primary output. Each PE in this example has 1 output and 2 inputs, from either the system input or a PE in the previous column.