Hardware genetic algorithm optimisation by critical path analysis using a custom VLSI architecture

Farouk Smith, Allan Edward van den Berg

Mechatronics Department, Nelson Mandela Metropolitan University, South Africa

ABSTRACT

This paper investigates optimisation of Evolutionary Hardware Systems (EHW) by means of digital circuit critical path analysis. A 2×2 digital multiplier and a Finite State Machine (FSM) control circuit were evolved using a target-independent Virtual Reconfigurable Circuit (VRC) architecture. An in-depth analysis of the phenotypes' Critical Paths (CP) was performed. Through analysing the CPs, it was shown that a great amount of insight can be gained into a phenotype's fitness. Particularly, the identification of the CP's dependence is valuable, since dependent CPs reduced the required net number of evolved Logic Elements (LE). Generally, in both the multiplier and state phenotypes, the CPs were evolved in ascending order of the net LEs. This suggests that evolution always favoured CPs with lower net numbers. However, we have seen that in one special case, if two independent CPs are used by a third CP, the resulting third CP has a lower net number than both independent CPs. The CP analysis also led to the development of the $Fitness_{Overall}$ fitness function, which had a distinctive way of not only rewarding correct output elements, but also encouraging more efficient evolution through sustaining evolved CPs, and further developing partially-evolved CPs. Finally, by using the optimized fitness function, we demonstrated the evolution of a FSM control circuit. The results verify that optimised GAs can find solutions quicker, and with fewer attempts.

KEYWORDS: FPGA, genetic algorithm, virtual reconfigurable circuit, evolutionary hardware, logic element, digital circuit, Cartesian genetic programming, hardware chromosome, intrinsic evolution, off-chip evolution, on-chip evolution

CATEGORIES: B.6.3 [**Logic design**]: Design aids—*automatic synthesis, hardware description language, optimisation, verification*

ARTICLE HISTORY

Received 2 September 2014
Accepted 24 March 2015

1 INTRODUCTION

Traditional circuit-design methodologies "rely on rules that have been developed over many decades" and require more human expertise for increasingly complex designs, which may be costly [1]. Complex designs are often tackled using powerful design tools, such as Electronic Design Automation (EDA), high-level abstraction design-techniques and advanced Internet Protocol core libraries. However, the design-productivity gap is still increasing [2].

One solution to this design problem is Evolutionary Hardware Systems (EHW). EHW is a combination of three disciplines: Computer Science, Electronics Engineering and Biology [3]. Through modelling biological and natural intelligence, engineers and scientists have been able to mimic natural evolution in software for use in hardware design [4].

Despite the increased research and resources in the field, EHW systems remain largely unusable in real-world applications [5]. Only a few engineering applications have shown promise [6, 7], even though early pioneers claim that evolution will soon be applied to large-scale machines [8].

Researchers have raised a number of issues that have retarded the growth of EHW applications. These include the difficulties in configuring EHW platforms, scalability [9], evolution time and problem complexity [10].

EHW systems make use of evolutionary algorithms (EA), usually genetic algorithms (GA), to evolve digital circuits on devices such as field-programmable gate arrays (FPGAs) [11, 12, 13]. These systems are particularly useful in adaptive control systems, fault-tolerant systems and the automatic design of low-cost hardware [4].

Email: Farouk Smith `Farouk.Smith@nmmu.ac.za`

In EHW, each individual is represented using a hardware chromosome, also called the genotype. A hardware chromosome is a string of integers that represents a certain circuit configuration when decoded. Each single integer in the hardware chromosome is referred to as a hardware gene.

The decoded circuit is known as the phenotype. Phenotypes are configured in terms of functionality (the function of each component) and routing (how the components are connected to one another).

Since EHW make use of genetic search algorithms that need to explore large search spaces, the intricacy and sheer scale of finding a solution becomes more apparent as the solution circuit becomes more complex. Many researchers have recognized that scalability is a hindrance in the successful implementation of EHW in real-world applications [14, 15].

Scalability, in this context, refers to the difficulty of finding a satisfactory solution for large complex problems, those found in real-world applications due to the GA's search space being too large, or the solution being too complex to be implemented on the EHW system. To put scalability in context, EHW systems have to evolve circuitry by placing digital logic gates, often thousands of logic gates in larger systems, in very specific configurations. In addition, as the complexity of the circuit increases, so the genotype length and the time required to calculate the fitness of each phenotype also increases. This results in there being billions of potential solutions which are cumbersome and time-consuming to explore, even with evolutionary techniques.

There are two EHW GAs investigated in this work to demonstrate fitness function optimisation in digital circuits to address the problem of scalability. The first GA variant, referred to as a canonical GA, uses the tournament-selection, uniform-crossover and mutation genetic operators. Initial studies on canonical GAs did not value mutation, and thus mutation did not often feature [16]. This is in direct contrast to the second GA variant, referred to as a $1 + \lambda$ GA.

The $1 + \lambda$ GA relies only on the mutation operator. It makes use of $(\mu + \lambda)$-selection, where μ represents the number of parents and λ the number of offspring. For example, a $1 + \lambda$ GA uses a single parent that is mutated λ times until a new generation is computed. This GA variant does not make use of tournament selection or crossover.

A noteworthy contribution of this research is the critical-path analysis of digital circuits in order to understand the manner in which they are evolved in EHW. With this knowledge, better fitness functions and operators were developed to enhance the EHW system's efficiency.

A further contribution of this research is the development of the 'overall fitness function', which has a distinctive way of not only rewarding correct output elements, but also encouraging more efficient evolution through sustaining evolved CPs, and further developing partially-evolved CPs.

2 RELATED WORK

2.1 Evolving multiplier circuits in hardware

A combinational-multiplier circuit is an electronic circuit that computes the product of two unsigned binary numbers. Multipliers are often useful for computing mathematical instruction-sets in PCs' arithmetic-logic units. Most multiplier circuits use the scheme of first computing the inputs' partial products, and then summing the partial products to form the final product.

The complexity of a multiplier's circuit increases exponentially with an increase in the number of output bits. For example, a conventional two-bit multiplier may use approximately eight FPGA logic elements configured to implement two half-adders, while a four-bit multiplier may use up to 64 logic elements configured to use four half- and eight full-adders. Considering the above complexity, the simple two-bit multiplier circuit is a favourable initial circuit to evolve since it small enough to demonstrate EHW while still being a practical sub-circuit for many digital ICs.

2.2 Scalability and chromosome length

Scalability and chromosome length are directly related. As the complexity (i.e. number of external IOs) of the solution circuit increases, so does the size of the search space and length of the chromosome string. Long chromosome strings are an inevitable side-effect of complex systems. Thus, by simplifying or scaling-down complex circuits, the chromosome strings can be reduced. This will reduce the EA's search space. However, a different approach to scalability is to increase the EA's computing power. To do this, researchers have proposed parallel evolution [17]. Parallel-evolution schemes make use of two or more independent EAs, i.e. the EAs run in parallel, thereby allowing multiple circuits to be evolved simultaneously.

2.3 Overcoming scalability using a multifaceted approach

In 2007, Wang et al. [17, p. 33] evolved both three-bit adders and multipliers in less than three seconds, which was reportedly "untouchable by any other reported evolvable system." The work showed that three-bit adders/multipliers were scalable if a mutifaceted approch was taken. Wang et al. suggested overcoming scalability using three techniques: optimising the EA; limiting the chromosome length; and "decreasing the computational complexity of the problem" [17, p. 25].

Firstly, Wang et al. made use of a GA optimised by omitting the crossover operator. Also, a multi-VRC platform was used, thereby allowing the GA to test the candidate circuits faster. Although not done by Wang et al., other optimisations could also include finding more effective mutation and crossover operators (if used), and improving the fitness function [18, 19].

Secondly, the chromosome length was limited by decomposing the solution circuit into modules, as done in modular evolution. There are different ways of decomposing circuits, each with varying levels of success and complexity. Examples include Shannon decomposition [20], disjunction decomposition [21], and output decomposition [17]. Wang et al. made use of output decomposition, which decomposes a circuit according to the number of external outputs.

Thirdly, to decrease the computational complexity, parallel evolution was used. Unlike modular evolution, parallel evolution does not decrease the complexity of the solution circuit. It only improves the computational capacity of the evolution. Wang et al. used a two-core system, with each core running independent GAs and VRCs, and evolving a single sub-circuit. Theoretically, there was no limit to the number of implemented cores and VRCs.

The results showed that that modular evolution decreased the number of generations required from over 18-million generations for standard evolution, to approximately 133 thousand. In addition, parallel evolution was able to improve the evolution time from approximately 77 seconds for standard evolution, to 2.6 seconds.

Finally, Wang et al. acknowledge that more complex circuits would still need to be tested, adding that "future work will be devoted to applying this scheme to other more complex real-world applications."

2.4 Variety of research

The diversity of EHW research may be considered both beneficial and problematic: beneficial because diversity promotes progress and unique solutions; problematic because there is little standardisation within the field.

To analyse the diverse EHW research, EHW systems can be classified, as suggested by Torresen [12], into the following categories: evolutionary algorithm, evolution level, target platform/architecture, degree of evolution and scope. The evolutionary algorithm and target platform was previously discussed. The remaining categories are defined below:

Evolution level: The level at which the evolution is performed is called 'granularity', with gate-level evolution being fine, and function-level evolution being course.

Fitness computation: Refers to the manner in which the fitness of a circuit is computed. Extrinsic evolution only downloads the elite chromosome to the target platform. Thus, much of the evolution is simulated. Intrinsic evolution implements and tests each chromosome in hardware.

Degree of evolution: Refers to whether or not "the evolutionary algorithm is performed on a separate processor incorporated into the chip containing the target EHW" [20]. Off-chip evolution does not make use of an incorporated processor, while on-chip evolution does. Complete evolution does not use a processor, but rather uses specialised hardware.

Scope: Static evolution only puts the evolved circuit to use once evolution is complete. Static evolution is typically used in Evolved Hardware. Dynamic evolution is undertaken while the evolved circuit is used. Thus, dynamic evolution is used in Evolvable Hardware.

Most research during the last 20 years deals with simple digital/analogue circuit synthesis [22], image filtering [16], and system refinements [23]. There is little evidence of real-world applications being implemented, and thus most research is still focussed on refining systems, and testing these refinements by evolving simple circuits such as adders and multipliers [17].

The EHW field was primarily target-platform driven, with the Xilinx 6200 FPGAs initially being the platform of choice. Later, post 2000, the Xilinx Virtex FPGAs limited direct-bitstream evolution, leading to the popularisation of VRCs, with many designers favouring VRCs due to cost and portability advantages.

Current work is concentrated on scalability; EHW needs to become more prominent in real-world applications. From the literature, there are many solutions to scalability, but the most promising solutions will need a multifaceted approach.

3 THE FITNESS FUNCTION OPTIMISATION

Variations in GAs—such as changing the population size, crossover, selection and mutation—may have a vast impact on the performance of the algorithm. GA operators are usually optimised for each EHW system according to the makeup of the evolutionary platform and architecture, the complexity of the circuit being evolved, the fitness function and the representation of the hardware chromosome. Thus, every GA is unique to the specific application.

This section introduces the VRC, GA operators, and hardware chromosome representation used to optimise the fitness function.

3.1 The virtual reconfigurable circuit era

The VRC is a second reconfigurable layer residing on top of an FPGA, which takes the form of a two-dimensional array of logic blocks, similar in architecture to Cartesian Genetic Programming (CGP) [24].

VRC-based solutions, such as those designed by Slorach & Sharman [25], Sekanina & Sllame [26], Sekanina & Freidl [11] and Smith [13], generally always consisted of:

Logic Elements (LEs): Programmable elements, sometimes called cells

Programmable interconnection network: Connecting the LEs and external IOs

Configuration memory: Stored the VRC's virtual bitstream so that the desired circuit could be implemented

The GAs in this work use extrinsic hardware evolution, by implementing the digital circuits on an FPGA by means of the VRC or Virtual FPGA designed by the first author, Smith [13].

In order to evolve the digital circuits, the VRC was setup to use 20 LEs, with 4 external inputs and 4 external outputs.

3.2 Reducing the search space

Evolution constraints can reduce the search space by lowering the possible routing and functionality permutations. For example, by limiting the LEs' functionality to fewer gates, the logic permutation can be reduced significantly. For this reason, each circuit's phenotype was limited to the following configurations, which were not permitted during evolution:

1. Each LE input was unique, i.e. an LE cannot have the same two inputs.
   Having identical inputs simply changes the LE's gate type, and is unnecessary.
2. Each LE's function was limited.
   The LEs' functionality was limited to the seven fundamental logic gates.
3. An external input cannot be directly connected to an external output.
   If connected directly, all the LEs are bypassed.
4. Each external output was unique.
   None of the external outputs can be connected to the same LE.
5. No feedback loops were allowed.
   Feedback loops can create memory elements within a circuit. This causes instability, thereby creating unreliable fitness values. For example, a circuit with feedback loops producing a fitness of $x\%$ during one evaluation may produce a completely different fitness value when tested again. This unstable circuit will cause inevitable genetic problems, since there is a high probability of the feedback loops being passed to the offspring.

   An LE array of m rows by n columns, which draws similarities from CGP, is used to prevent feedback loops [11, 24]. An LE's output can only be connected to an external output or the input of another LE which is in a following column. If an LE's output is connected to a preceding column's LE input, there is a risk of creating a feedback loop.
6. External inputs could only be connected to column-zero LEs.
   In order to further reduce the GA's search space, the external inputs could only be connected to the inputs of LE's in column zero.

3.3 Testing a chromosome

During each GA generation, every chromosome needs to be tested. Testing is done by comparing a circuit's output to a truth table modelled on the desired circuit.

During testing, each input vector is sequentially loaded onto the external inputs of the VRC, and the resultant external output vector is recorded. Fig. 1 shows input vector $[0, 0, 0, \ldots, 1, 0]$ being loaded onto the VRC external inputs. The resultant output vector, which in this arbitrary example is $[0, 1, \ldots, 1, 1]$, can then be compared to the corresponding output vector from the desired circuit's truth table.

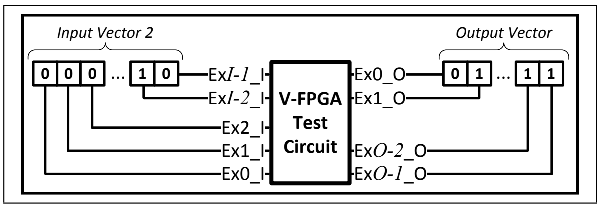


Figure 1: Loading an input vector onto the external inputs

3.4 Fitness calculation

To explain how a phenotype's fitness is calculated, first consider the conventional multiplier-circuit in Fig. 2.

Each external output has its own critical path (CP). A CP is the direct path linking the external inputs to a particular external output. Hence, the multiplier circuit can be thought of as four, smaller CP circuits that have been coupled.

CPs may be classified as either independent or dependent. An independent CP does not have any LEs in common with other CPs. For example, C0's CP is independent since none of its LEs are used by any other CP (highlighted red in Fig. 2). Dependent CPs, such as those of C1, C2 and C3, have LEs in common.

Now, consider the multiplier's truth table, shown in Fig. 3. There are three sets of data within the truth table that can be used to derive a phenotype's fitness, namely:

- The 16 output vectors of the corresponding test vectors
- The 64 individual output elements of the output vectors
- The 4 CP vectors of each external output

First, consider the output vectors. In initial evolution trial-runs, the phenotypes were awarded fitness values according to Equation 1, which expresses the percentage of correct output vectors. For example, if a phenotype had 12 correct output vectors, it would score an output-vector fitness of

$$F_{OV} = \left(\frac{\text{Correct output vectors}}{\text{Output vectors}}\right)\%$$

Equation 1: Fitness function of the output vectors

However, this fitness scheme is flawed when applied to digital logic circuits. Consider the AND-gate marked as callout A in Fig. 2. This gate only affects the output of C_3. If the gate's function was arbitrarily changed, most of C_3's outputs would be incorrect. This would, in turn, lead to most output vectors also being incorrect, thereby yielding a low fitness value.

Nevertheless, this low fitness value would actually be underrated; since even though the phenotype's output vectors are mostly incorrect, the phenotype's LEs and routing is generally correct (only the changed gate is incorrect). For this reason, assigning fitness values using the output vectors was considered inaccurate and misleading.

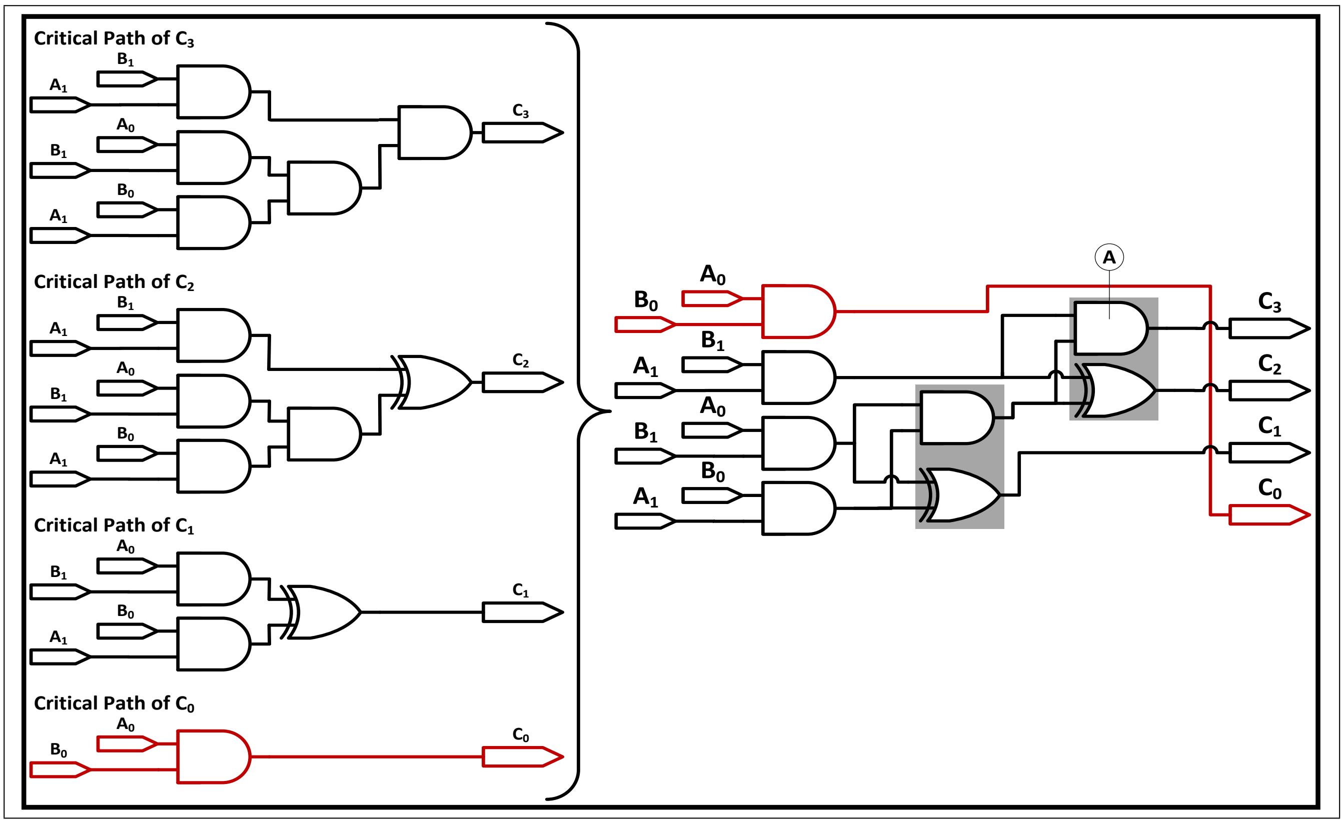


Figure 2: Conventional multiplier circuit (right) comprising of four smaller CP circuits (left)

A more useful and precise fitness value was derived from the output elements and CP vectors.

There are $O(2^I) = 4(2^4) = 64$ output elements for O external outputs and I external inputs. Equation 2 expresses the number of correct elements as a percentage. Thus, if an example phenotype had 24 of the 64 elements correct, a fitness of $F_{Elements} = \frac{24}{64} = 37.5\%$ would be assigned.

$$F_{Elements} = \left(\frac{\text{Correct output elements}}{\text{Output elements}}\right)\%$$

Equation 2: Fitness function of the output elements

The advantage of using output elements over output vectors is that every correct element in the output vectors contributes towards the final fitness value. For example, if all the C_3 output elements in Fig. 3 were incorrect, the truth table would yield zero correct output vectors but 16 incorrect output elements. Thus, the phenotype would attain a fitness value of $F_{Elements} = \frac{48}{64} = 75\%$ for an element evaluation, but $F_{OV} = 0\%$ for the output-vector evaluation. The output-element evaluation would provide a more accurate fitness value since the phenotype is partially correct.

$F_{Elements}$ does not, however, encourage the correct evolution of CPs. It merely gives an overall indication of a phenotype's correctness.

To understand why the correct evolution of the CPs is important, note that CPs are often dependent. Because dependent paths rely on other CP LEs, correctly evolving one CP inevitably partially solves other CPs. For example, if C_1's CP in Fig. 2 was to be successfully evolved, two of the five LEs in paths C_2 and C_3 would, by default, also be correct. This would in turn make the GA more efficient.

To encourage CP evolution, the CP vectors in Fig. 3 need to be evaluated. To do this, the CP fitness, or F_{CP}, is used to express the percentage of correct CP vectors, as shown in Equation 3. For example, if a phenotype has three correct CP vectors, it would score a CP-vector fitness of $F_{CP} = \frac{3}{4} = 75\%$.

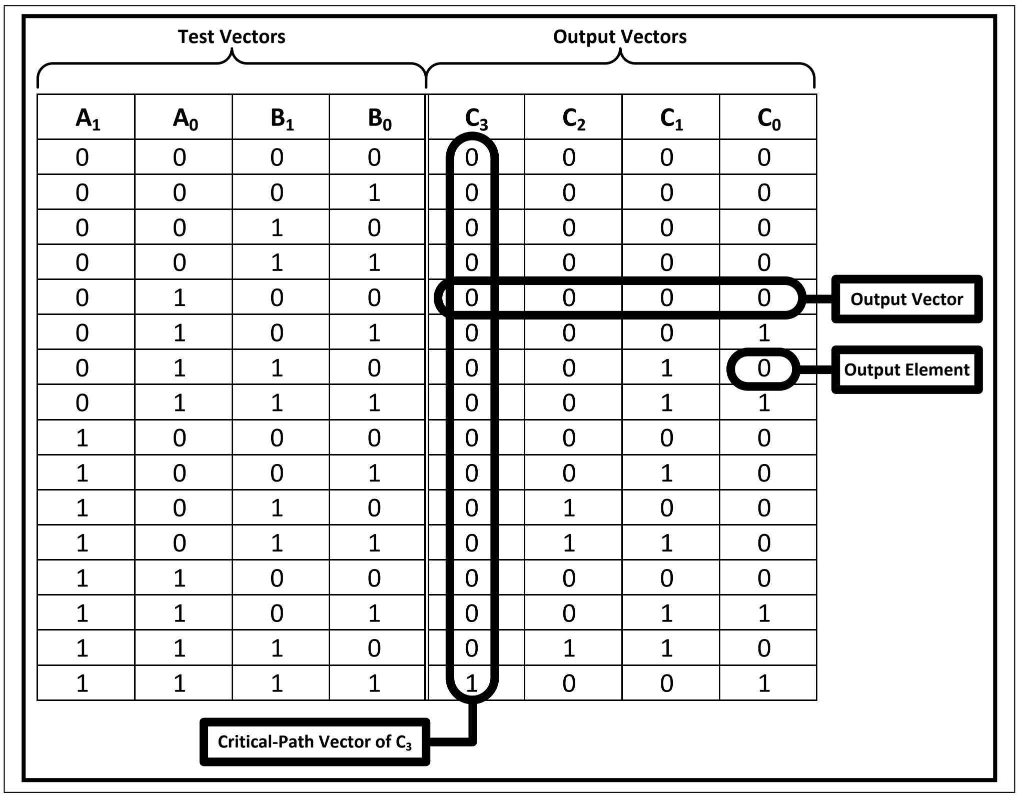
$$F_{CP} = \left(\frac{\text{Correct critical path vectors}}{\text{Critical path vectors}}\right)\%$$

Equation 3: Fitness function of the CPs

The problem with equation 3 is that F_{CP} is very rigid, and only rewards fully evolved CP vectors, partially evolved CPs are not rewarded. For partially evolved CPs, Equation 4 is used.

To explain Equation 4, first consider each CP vector in Fig. 3:

- C_0's CP vector requires 4 true bits and 12 false bits
- C_1's CP vector requires 6 true bits and 10 false bits
- C_2's CP vector requires 3 true bits and 13 false bits



Test Vectors				Output Vectors			
A_1	A_0	B_1	B_0	C_3	C_2	C_1	C_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0
0	1	0	1	0	0	0	1
0	1	1	0	0	0	1	0
0	1	1	1	0	0	1	1
1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0
1	0	1	0	0	1	0	0
1	0	1	1	0	1	1	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	1	1	0
1	1	1	1	1	0	0	1

Output Vector
Output Element
Critical-Path Vector of C_3

Figure 3: Analysis of a 2-bit multiplier's truth table

$$F_{CP,Partial} = \left(\frac{\sum_{i=0}^{3} 0.5(\text{of correct } \textit{true} \text{ bits for } C_i'\text{s CP vector}) + 0.5(\text{of correct } \textit{false} \text{ bits for } C_i'\text{s CP vector})}{\text{Critical path vectors}} \right)\%$$

Equation 4: Fitness function of the partial CPs

- C_3's CP vector requires 1 true bit and 15 false bits

For each CP vector, the percentage of correct true and false bits is calculated, and weighted in a 0.5:0.5 ratio in Equation 4. For example, if C_0's CP vector yielded 3 correct true and 10 correct false bits, a fitness of $0.5(\frac{3}{4}) + 0.5(\frac{10}{12}) = 79.2\%$ would be assigned. Once all four CP-vectors have been assessed, the mean of the four CP-vector fitness values can then be expressed as $F_{CP,Partial}$.

An important aspect of Equation 4 is the 0.5:0.5 ratio, in which the percentage of correct true and false bits are weighted. This ratio is essential. If not used, simply finding the percentage of correct bits will yield inaccuracies. To explain, consider the circuit in Fig. 4, which will always register a logic low regardless of the input signals. If, for example, this circuit was evolved as C_3's CP, the CP vector would register 16 false bits. Thus, the CP-vector fitness evaluation would identify 15 of these false bits as correct and only one as incorrect.

However, these 15 correct bits are deceptive. From Fig. 2, note that C_3's CP is one of the multiplier's most complex CPs and makes use of five LEs. When comparing the desired C_3 CP circuit in Fig. 2 to the evolved circuit in Fig. 4, there is a substantial difference.

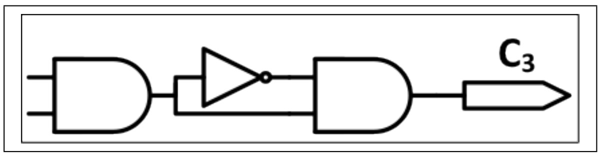


Figure 4: Logic-low circuit

Thus, assigning a CP-vector fitness value of $\frac{15}{16} = 93.8\%$ will result in an overrated fitness score, since the evolved CP does not resemble the desired CP.

One way to curb this inaccuracy is to place an equal amount of emphasis on both the true and false bits. This is what the 0.5:0.5 ratio does. If this ratio were to be applied to the above example, a fitness of $0.5(\frac{0}{1}) + 0.5(\frac{15}{15}) = 50\%$ would be achieved. This lowered fitness value is more truthful, as it more fittingly describes the poorly evolved CP circuit.

Finally, all three fitness values, namely $F_{Elements}$, F_{CP} and $F_{CP,Partial}$ could now be combined in order to describe the phenotype's overall fitness, as shown in Equation 5. Both $F_{Elements}$ and $F_{CP,Partial}$ were given an equal weighting of 30% of the overall fitness. However, to ensure that fully evolved CPs are preserved during the evolution process, a slightly higher 40% weighting was given to the F_{CP} fitness.

In summary: $F_{Elements}$ provides an overview of a

$$Fitness_{Overall} = (0.3F_{Elements} + 0.4F_{CP} + 0.3F_{CP,Partial})\%$$

Equation 5: Overall fitness function

phenotype's correctness, F_{CP} ensures that correct CPs are sustained, and $F_{CP,Partial}$ encourages the correct evolution of partially evolved CPs.

4 RESULTS

The results will be addressed in two parts: the outcome of the canonical evolution and the outcome of the $1+\lambda$ evolution.

In order to aid comparison, the following control variables were kept constant:

- Both GAs made use of a six-individual population
- Both GAs made use of the same fitness function (Equation 5)
- Both GAs made use of the same GA constraints (Section 3.2)

4.1 Canonical evolution

The canonical-evolution results are presented in two subsections below. The first section discusses the results of an initial trial run; the second section discusses the final-canonical GA results.

4.1.1 Trial-canonical results

The first successful-trial GA to evolve a 100%-fit phenotype made use of fifty individuals; and all LE functionality was allowed, unlike the constraint specified in Section 3.2, point 2, i.e. 16 LE functions were permitted. (This is in direct contrast to the final-canonical GA's parameters, which used a six-individual population and only fundamental LEs.) Furthermore, another notable difference was the trial's fitness function, shown in Equation 6. The function does not consider the CP vectors.

$$Fitness_{Trial} = (0.75F_{Elements} + 0.25F_{OV})\%$$

Equation 6: Trial fitness function

Nevertheless, even though the trial GA did not use optimised parameters, it was successful.

Fig. 5 shows the progress of the fittest individual using the successful-trial GA. The first parent had an initial fitness of 51.5%, which steadily increased during the first 50 generations. Notice that there are large jumps in the graph. This is expected, since changing the routing or function of one LE can dramatically improve, or deteriorate, a phenotype's fitness.

The following milestones are also noted in Fig. 5's graph:

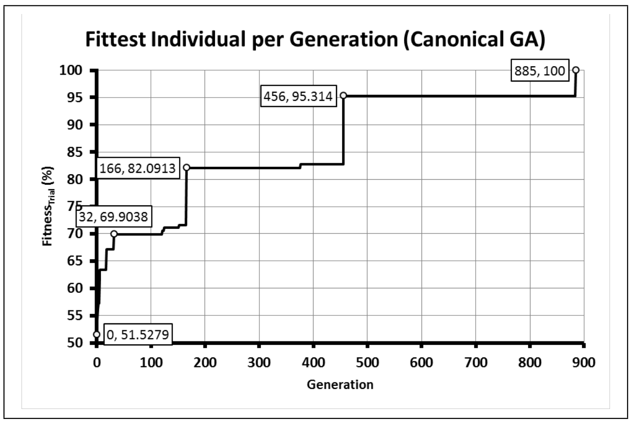


Figure 5: The results of the successful trial

- At fitness values 69.9%, 82.1%, 95.3% and 100%, the outputs C_0, C_3, C_1 and C_2 respectively are correctly evolved. C_0's CP passes two LEs, C_3 and C_1's CPs pass three LEs, while C_2's CP passes six LEs.
- It took a total of 885 generations to evolve the phenotype.

Fig. 6 shows the evolved phenotype. The bold circuitry shows the CPs. Notice that of the twenty available LEs, only ten were used in the evolved circuit.

Fig. 6 can be further simplified into the circuit shown in Fig. 7 by removing:

- The wire LEs, i.e. LEs which only pass data through them. There are two examples of wire LEs in Fig. 6, which have been demarcated as callouts A.
- Redundant NOT-gates. The NAND-gate (callout B) is connected to a NOT-gate (callout C) and an inverted input to an AND-gate (callout D). Thus, the two bubbles and NOT-gate are, in reality, redundant and can be removed.

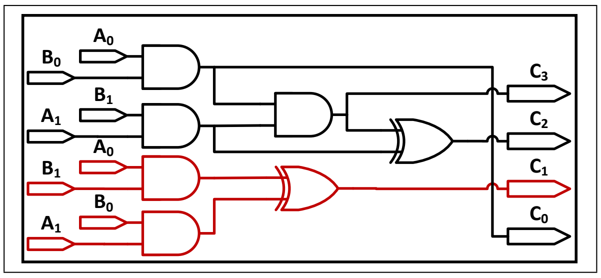


Figure 7: Simplified trial phenotype (red outline indicates an independent CP)

4.1.2 Final-canonical results

After analysing the successful trial runs, the importance of rewarding correctly evolved CPs was con-

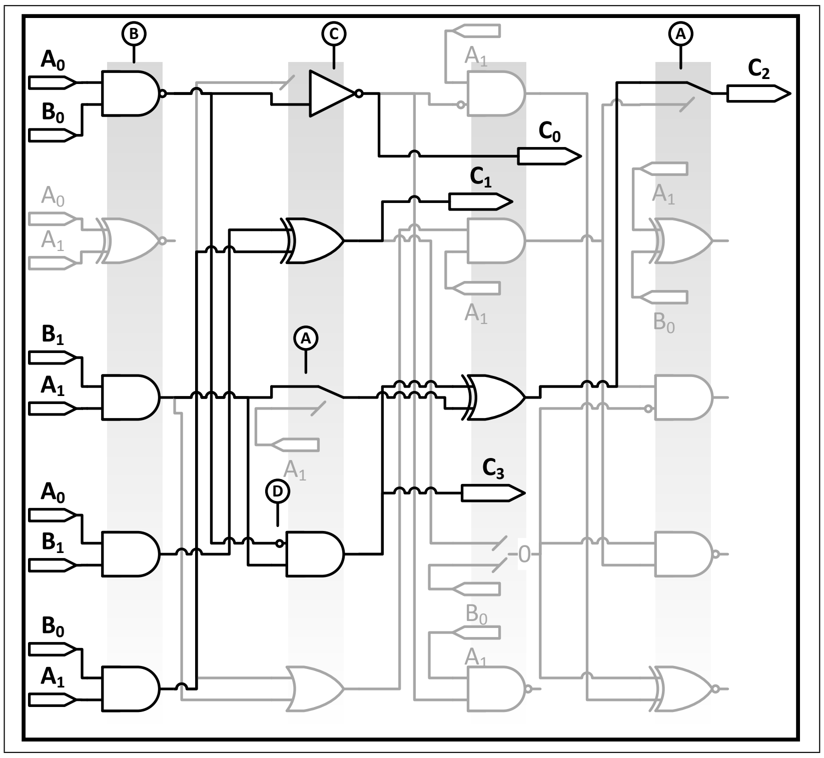


Figure 6: Evolved trial phenotype

firmed. This in part led to the derivation of the $Fitness_{Overall}$ fitness function (Equation 5).

The final-canonical GA was executed eleven times before a 100%-fit phenotype was evolved. Each run was limited to 3000 generations.

Fig. 8 shows the progress of the fittest parent using the final-canonical GA. Within the first 200 generations, the GA had evolved two of the four CPs. As the case with the trial-canonical GA, notice that there are large jumps in the graph. However, unlike in the trial, these jumps are largely due to the F_CP function in the $Fitness_{Overall}$ fitness evaluation. When a correct CP is evolved, the F_CP variable increases the overall fitness by 10%, resulting in noticeable jumps.

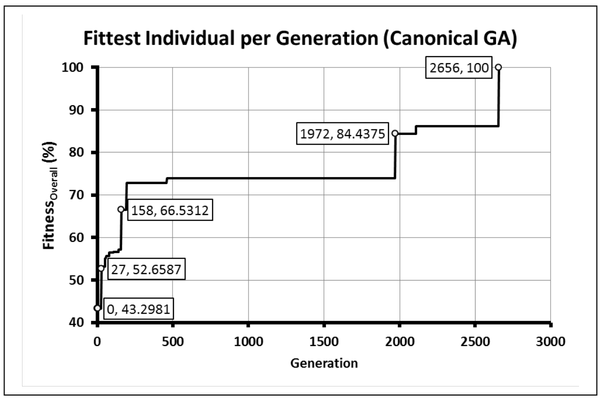


Figure 8: Results of the final-canonical GA

The following milestones are also noted in Fig. 8:

- At fitness values 52.7%, 66.5%, 84.5% and 100%, the outputs C_0, C_3, C_2 and C_1 respectively are correctly evolved. C_0's CP passes one LE, C_3 and C_1's CPs pass three LEs, while C_2's CP passes five LEs.
- It took a total of 2656 generations to evolve the phenotype.

The final-canonical phenotype is shown in Fig. 9, with the bold circuitry showing the CPs. Out of the twenty available LEs, nine were used.

Fig. 9 can be further simplified into the circuit shown in Fig. 10 by removing the wire LEs. Since the final-canonical GA only made use of fundamental gates, there are no redundant bubbled or NOT-gates.

4.2 $1 + \lambda$ evolution

The $1+\lambda$ GA was executed eight times before a 100%-fit phenotype was evolved. As the case with the canonical GA, each run was limited to 3000 generations.

Fig. 11 shows the progress of the fittest individual using the $1+\lambda$ GA. Notice that there are four spikes in the graph (demarcated with $\times$). These spikes represent downloading errors, where the phenotype has been incorrectly downloaded onto the V-FPGA.

The following milestones are noted in Fig. 11's graph:

- At fitness values 55.5%, 70.0%, 82.2% and 100%, the outputs C_0, C_1, C_3 and C_2 respectively are correctly evolved. C_0's CP passes one LE; C_1's CP passes three LEs; C_3's CP passes five LEs; while C_2's CP passes nine LEs.
- It took a total of 1711 generations to evolve the phenotype.

The final $1 + \lambda$ phenotype, shown in Fig. 12, made use of eleven LEs. Of these eleven, three were wire LEs.

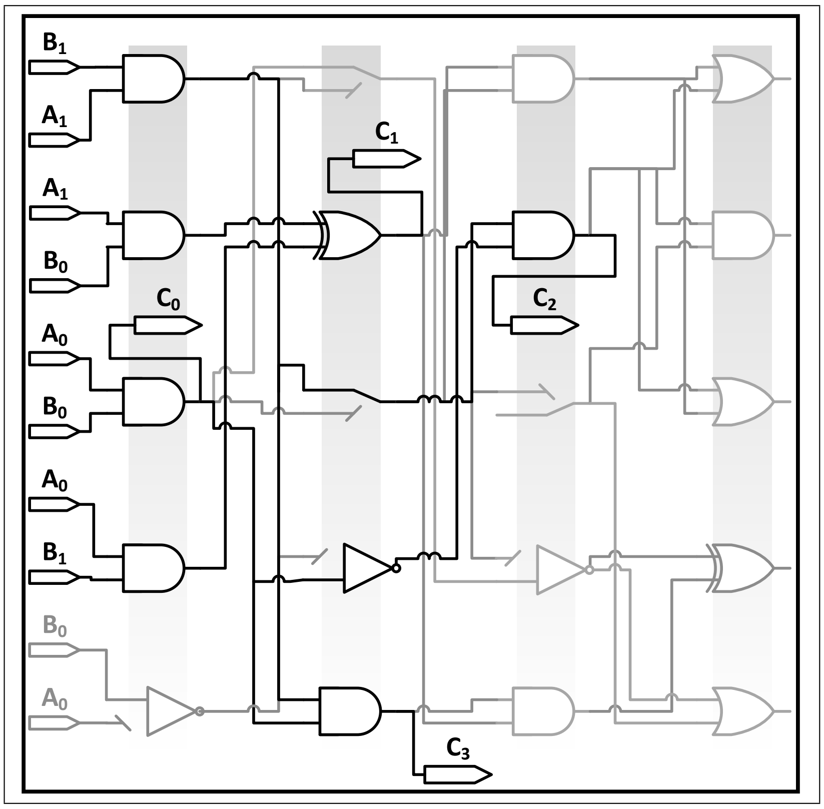


Figure 9: Canonical phenotype

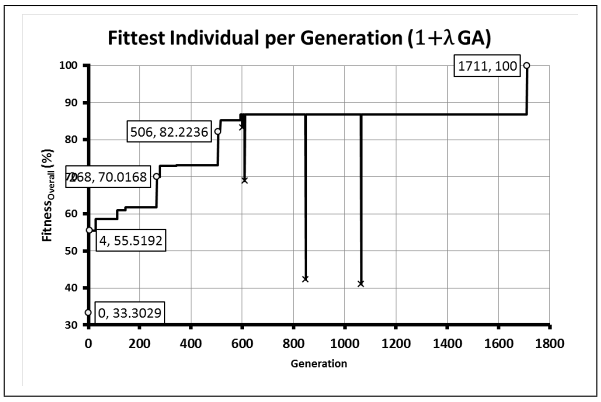


Figure 11: Results of the $1 + \lambda$ GA

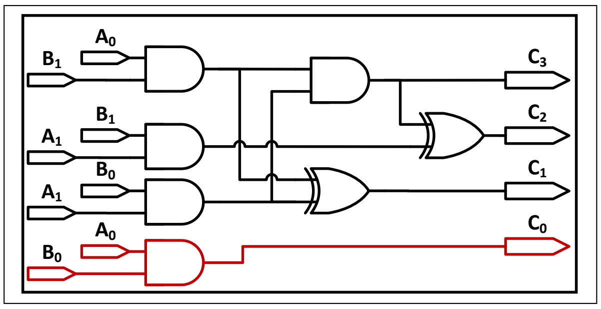


Figure 13: Simplified $1 + \lambda$ phenotype (red outline indicates an independent CP)

Again, the wire LEs was removed in the simplified circuit, shown in Fig. 13. C_0's CP is the only independent CP.

5 DISCUSSION

5.1 The genetic algorithms

Comparing the final-canonical and $1 + \lambda$ results, the final-canonical GA took 945 generations longer to find a solution. In addition, the $1 + \lambda$ GA found a solution after eight attempts, compared to the final-canonical GA's eleven.

Although the study was based on a small sample of evolution attempts, overall, the results suggest that the $1 + \lambda$ GA was more efficient, and agree with the findings of Vassilev et al. [19] and Sekanina & Freidl [11]. The subsections below discuss possible reasons why.

5.1.1 The crossover and mutation operators

In traditional GAs, used to solve mathematical problems, the chromosomes are represented using floating-point notation. The crossover operator creates new offspring by combining the parents' genes. This is done, for example, by finding the arithmetic mean of each pair of genes. The result is that the offspring's fitness is never worse than a parent's fitness.

The above idea falls under the topic of 'evolvability'. Evolvability is defined as the ability for an EHW system to produce individuals fitter than those found in previous generations [27]. To examine why EHW systems have poor evolvability, we need to consider the GAs' fitness landscapes.

For traditional GAs, the fitness landscape is considered to be smooth, resulting in the offspring always converging towards a solution. However, this is untrue in EHW systems.

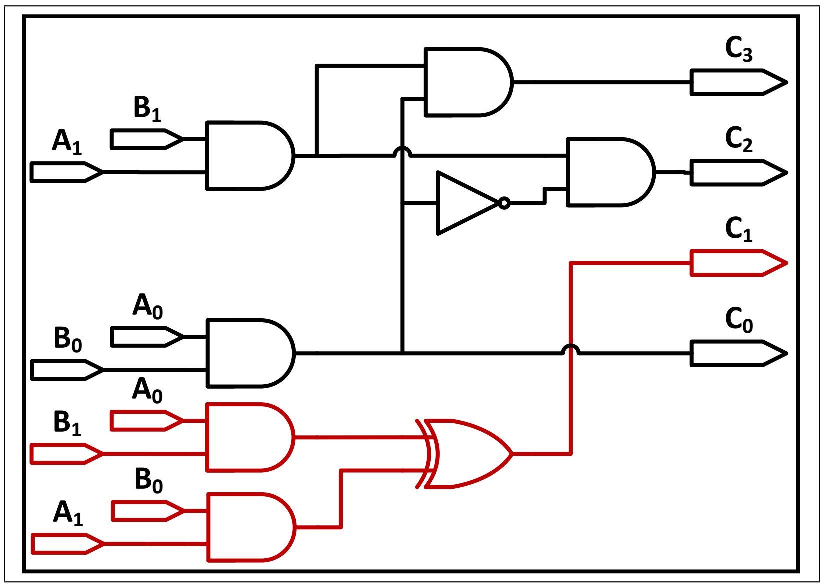


Figure 10: Simplified canonical phenotype (red outline indicates an independent CP)

EHW systems have rugged fitness landscapes, where small changes in a gene dramatically influence the fitness. For example, one altered gene can map a NOT-gate in a fit phenotype, thereby inverting all the output signals and completely destroying the phenotype. Similarly, simple routing changes can also influence the fitness of a phenotype.

Vassilev et al. [19] elaborate:

> The difference [between a smooth and rugged landscape] originates in the structure of the genotypes, which are strings defined over two completely different alphabets, and are responsible for the functionality and connectivity of the array of logic cells.

Stated differently, there is no mathematical correlation between the phenotype's fitness and the genotype's logic and/or routing genes.

During evolution, two parents may have similar fitness values, but their phenotypes can be completely dissimilar. This raises concerns as to how to implement crossover, if at all. Simply swapping the parents' genes, randomly combining segments of two different parent phenotypes' topologies and functionality, will inevitably result in weak offspring.

Thus, to maintain a system's evolvability, crossover should be used with caution when applied to digital circuitry. Instead, as used in the $1 + \lambda$ GA, an EHW GAs should rely on mutation. By making small adjustments to a phenotype, there is a greater probability of producing fit offspring.

5.1.2 The population size

In the initial trials, large populations, with fifty or more individuals, were used. These large populations inevitably took longer to execute since there were more individuals to evaluate.

However, it was found in the final-canonical and $1 + \lambda$ GAs that large populations were unnecessary. To explain why, first consider a fifty-individual population. To create a new generation, the fittest parent in the population is crossed-over and/or mutated fifty times according to the implemented GA. This means that even if the first offspring is fitter than the parents, the GA will continue to crossover/mutate the parents from the original population until fifty new offspring are created. Only once the new generation is formed will this fitter offspring become the new parent.

Now, consider a smaller six-individual population, where the GA crosses-over/mutates the parent six times. Unlike in the large population, if the first offspring is fitter, the GA only has to create five more offspring (and not 49) in order to form a new six-individual population. Again, once complete, this fitter offspring will become the new parent.

Because smaller populations are evaluated in smaller batches, the fittest parents are updated more regularly than in larger populations. This ensures the mutation and crossover operators are more effective and the GA has a greater level of efficiency. Thus, the smaller six-individual population was favoured.

5.2 The evolved phenotypes

5.2.1 Comparing the simplified evolved phenotypes

The conventional and evolved multiplier circuits show similarities in that they all made use of the same external-input combinations to the AND-gates. These AND-gates are crucial since they calculate the partial products of the multiplicand and multiplier. However, unlike in the conventional circuit which adds the partial products, none of the evolved phenotypes made use of the half-adders.

All three simplified phenotypes are unique, showing that there is more than one solution circuit within a GA's search space. In particular, the final-canonical phenotype is interesting because it did not evolve a second XOR-gate (as found in the trial-canonical and $1+\lambda$ phenotypes). This uniqueness is a by-product of the inherent degree of randomness a GA possess, as seen in

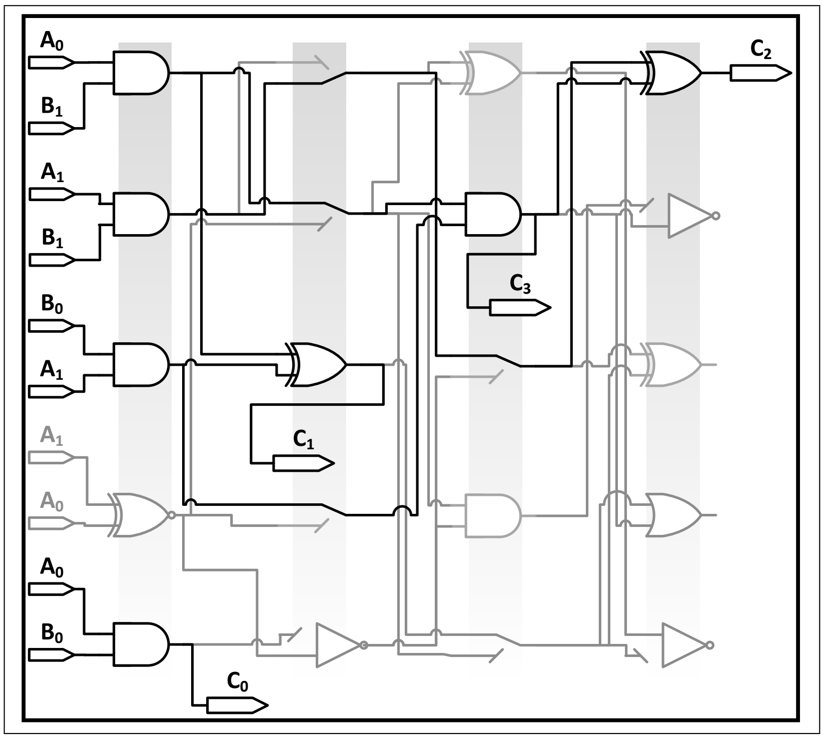


Figure 12: $1 + \lambda$ phenotype

the random mutation, crossover and initial population. Also, the uniqueness demonstrates a major advantage of using an EHW system, they can autonomously find unusual, novel and often more efficient solutions to problems.

To show that the evolved phenotypes are often more efficient, consider Table 1. The table summarises the total number of LEs used by each simplified phenotype, as well as the number of LEs used by each CP.

Table 1: LE summary of the simplified phenotypes

	C_0	C_1	C_2	C_3	Total no. of LEs
Conventional multiplier	1	3	5	5	8
Trial-canonical phenotype	1	3	4	3	7
Final-canonical phenotype	1	3	4	3	8
$1 + \lambda$ phenotype	1	3	5	3	7

From the table, the following is observed:

- C_0 and C_1's CP remained unchanged in both the evolved and conventional circuits.
- C_2's CP in the conventional multiplier used five LEs. This was improved upon in both canonical phenotypes by using only four LEs, but remained unchanged in the $1 + \lambda$ phenotype.
- Again, C_3's CP in the conventional multiplier used five LEs. This was improved upon in all evolved phenotypes by using only three LEs.
- Both the conventional circuit and final-canonical phenotype made used of eight LEs, while the trial-canonical and $1 + \lambda$ phenotypes only used seven.

In summary, of the four evolved CPs, two remained unchanged; one was usually improved upon; while one was always improved. No phenotype was less efficient than the conventional circuit, with two of the three evolved phenotypes improving the circuit's efficiency by one LE. All evolved phenotypes were unique.

5.2.2 The evolved critical paths

In all the experiments, using both the canonical and $1+\lambda$ GAs, the sequence in which the CPs were evolved was also unique. For the three successfully evolved phenotypes, the sequence of the evolved CPs was as follows:

- Trial-canonical GA: C_0 C_3 C_1 C_2, where C_1 is independent
- Final-canonical GA: C_0 C_3 C_2 C_1, where C_1 is independent
- $1 + \lambda$ GA: C_0 C_1 C_3 C_2, where C_0 is independent

The above CP results are summarised in Table 2:

1. The first row in the table shows the number of LEs used by each CP.
2. During evolution, due to some CPs being dependent, some LEs are shared and thus only need to be evolved once. This is shown in the second row. Thus, all independent CPs and the first evolved dependent CPs will always have no previously evolved LEs, and will yield a 0 in the second row of the table.
3. Finally, by finding the difference between the number of used LEs and the number of previously evolved LEs, the net number of LEs that was

needed to be evolved for the particular CP can be calculated, as shown in the final row.

From the table, the following is observed:

- C_0's CP was always evolved first, regardless of its dependence. This is due to its simplicity, i.e. it only used one or two LEs.
- C_1's CP sequence varied–from third to fourth to finally second place.

In both canonical GAs, C_1 was independent and thus was never partially evolved with the evolution of the other CPs. This explains why the CP took longer to evolve when compared to the $1 + \lambda$ phenotype.

For the $1 + \lambda$ phenotype, C_1's CP was the first dependent CP to be evolved. This is due to its simplicity when compared to the other dependent CPs in the $1 + \lambda$ phenotype, i.e. it used three LEs compared to the five and nine LEs used by C_3 and C_2 respectively.

- C_2's CP was evolved either third or last, mostly due to its complexity. In all three phenotypes, C_2's CP used the most LEs–it used five LEs in the final-canonical phenotype, six LEs in the trial-canonical phenotype and nine LEs in the $1 + \lambda$ phenotype. However, due to C_2's CP always being dependent, the net number of evolved LEs was much lower, and thus C_2's CP evolved in a reasonable amount of time. This is particularly evident in the $1 + \lambda$ GA, where C_3's CP is a key component to C_2's CP, providing five of the nine required LEs.
- Although C_3's CP was evolved either second or third, it was always the second dependent CP to evolve due to it neither being the simplest, nor the most complex dependent CP.

The net-evolved-LE number reveals an important insight into the manner in which a GA evolves the phenotypes. Note that the net number, for a particular phenotype, increases for each CP. This shows that GAs tends to evolve CPs with smaller net numbers first. Thus, the fewer net LEs a CP requires, the more likely a GA will correctly evolve the CP. This is expected, since intuitively there is a higher probability of correctly evolving a simpler CP which uses fewer LEs.

5.2.3 Erroneously-evolved chromosomes

During evolution, as more CPs are successfully evolved, so the probability of mutating a correct LE increases; and the number of available LEs decreases. This implies that as individuals become fitter, so the difficulty finding a solution also increases.

For example, suppose for the 20-LE VRC, one CP is correctly evolved using four LEs, with the remaining sixteen LEs yet to be evolved for the other CPs. One may think that the GA now has a greater chance of success since there are fewer routing or logic configurations, i.e. there are only sixteen potential LEs to be evolved compared to the initial twenty. However, this is not the case, since these four evolved LEs can still be altered by the GA. During evolution, the GA cannot distinguish between correctly and incorrectly evolved LE. Consequently, the GA can modify any LE, even if correct. Thus, in actuality, there is an increased chance of erroneously altering a correct LE, thereby making it less likely to successfully evolve the next CP. The above explanation is reflected in the results. Most failed attempts, managed to evolve three of the four CPs, with the complex or independent CP failing to evolve.

To further clarify the above explanation, consider another example. Suppose a GA has correctly evolved three of the four CPs. During evolution, even if the GA correctly evolves the fourth CP in a particular phenotype, there is a high probability that the GA, in the process of evolving this fourth CP, will erroneously alter the other three CPs. Thus, the fitness function will return a low value for this phenotype since one or more of the original three CPs are now incorrect.

It may be beneficial for a GA to identify and isolate correct genes within a chromosome. By doing this, there will be no chance of erroneously modifying correct LEs, and thus mutation and crossover will only be applied to the genes still requiring further evolution.

6 IMPLEMENTATION OF A FINITE-STATE MACHINE (FSM)

A FSM's control circuit, represented in Fig. 14, used in a typical mechatronics application, needs to be designed and evolved. To do this, the system's control requirements, components and operation are first defined. Then, based on these parameters, the FSM is modelled using a state diagram. Based on the FSM, we derived the next state and output tables, which later forms a core part of the GA's fitness function. Finally, by using the hardware VRC setup and GA discussed in the previous section, each state's combinational sub-circuit is then independently evolved.

6.1 Case study

A packaging company, which manufactures corrugated boxes, makes use of a FSM control circuit that controls the production of glue in two tanks. The first tank, the mixing tank, is used to mix starch and water together, at a specified temperature, in order to produce a batch of glue. The predefined temperature set-point is selected by the tank's operator via a numeric keypad. This set-point determines the glue's viscosity, an important aspect influencing the integrity of the final box. After mixing and heating, the glue is then pumped into a second tank, the holding tank.

The holding tank stores the glue, also at a specific temperature, until it is needed by the factory's gluing machinery. When the glue is pumped from the holding tank, it is pumped in a ten-second-on, five-second-off cycle. The pumped glue is used to produce the board needed for the boxes.

From the system description, the tanks' operation was analysed and modelled as a FSM, using two states, the mixing-tank and holding-tank states.

Table 2: Summary of the net number-of-LEs evolved by each CP (red text indicates an independent CP)

	Trial-canonical phenotype				Final-canonical phenotype				$1+\lambda$-phenotype			
	C_0	C_1	C_2	C_3	C_0	C_1	C_2	C_3	C_0	C_1	C_2	C_3
Number of evolved LEs used by the CP	2	3	3	6	1	3	5	3	1	3	5	9
Number of LEs used by the CP that were previously evolved by other CPs	0	1	0	3	0	1	2	0	0	0	2	5
Net number of LEs that were evolved (Net evolved LEs)	2	2	3	3	1	2	3	3	1	3	3	4

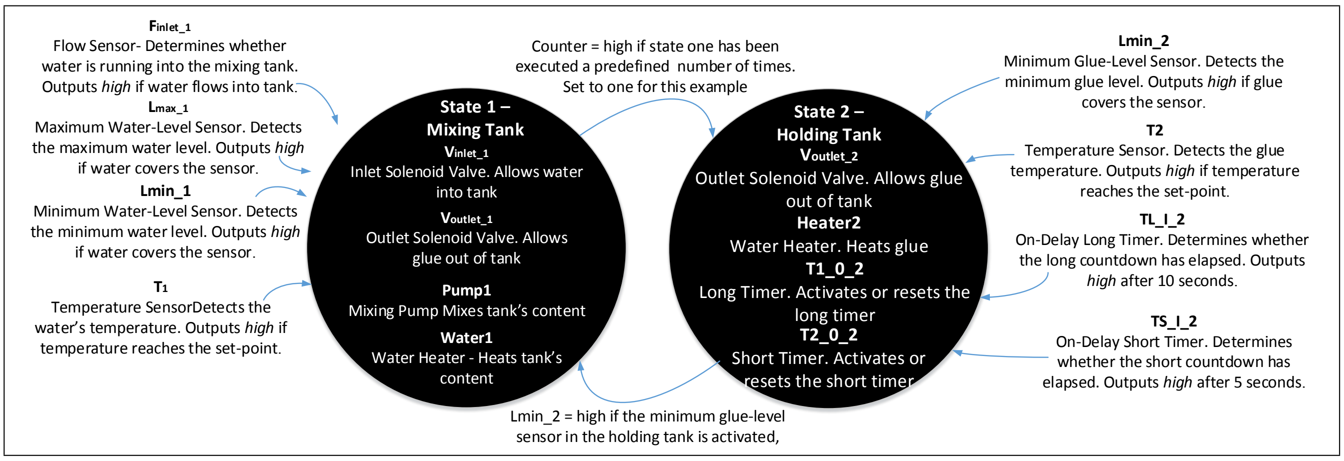

Finlet_1
Flow Sensor- Determines whether water is running into the mixing tank. Outputs *high* if water flows into tank.
Lmax_1
Maximum Water-Level Sensor. Detects the maximum water level. Outputs *high* if water covers the sensor.
Lmin_1
Minimum Water-Level Sensor. Detects the minimum water level. Outputs *high* if water covers the sensor.
T1
Temperature SensorDetects the water's temperature. Outputs *high* if temperature reaches the set-point.
Counter = high if state one has been executed a predefined number of times. Set to one for this example
State 1 – Mixing Tank
Vinlet_1
Inlet Solenoid Valve. Allows water into tank
Voutlet_1
Outlet Solenoid Valve. Allows glue out of tank
Pump1
Mixing Pump Mixes tank's content
Water1
Water Heater - Heats tank's content
State 2 – Holding Tank
Voutlet_2
Outlet Solenoid Valve. Allows glue out of tank
Heater2
Water Heater. Heats glue
T1_0_2
Long Timer. Activates or resets the long timer
T2_0_2
Short Timer. Activates or resets the short timer
Lmin_2 = high if the minimum glue-level sensor in the holding tank is activated,
Lmin_2
Minimum Glue-Level Sensor. Detects the minimum glue level. Outputs *high* if glue covers the sensor.
T2
Temperature Sensor. Detects the glue temperature. Outputs *high* if temperature reaches the set-point.
TL_I_2
On-Delay Long Timer. Determines whether the long countdown has elapsed. Outputs *high* after 10 seconds.
TS_I_2
On-Delay Short Timer. Determines whether the short countdown has elapsed. Outputs *high* after 5 seconds.

Figure 14: State diagram of the tank system

6.2 State diagram

Fig. 14 shows a basic state diagram of the tank system with three important sets of data:

1. The sequence in which the states are executed
2. The conditions for the current state to be executed
3. The conditions to transition from one state to the next

Based on the FSM, we derived the next state and output tables for each state. The system consists of five sub-circuits: combinational logic, sequential logic, counter circuit and two timing circuits.

To evolve the FSM's combinational circuit, a truth table used by the fitness function, representing the outputs for each possible external-input combination, was required for each state. Two separate combinational circuits, one for each state, were then evolved. The truth table used to control the mixing tank's inlet and outlet valves, heater and pump, as well as the holding tank's truth table, were derived similarly from the FSM.

The combinational logic portion, implemented using Boolean logic, connects directly to the system's sensors and actuators. It controls the system's external outputs according to the logic combination on the external inputs, timer inputs and state lines.

Following on from the previous section, the same EHW setup, used to evolve the multiplier, was used to evolve the combinational logic of the FSM. In addition, the GA made use of the same evolutionary constraints, elitism, mutation and $Fitness_{Overall}$ function (from Section 3.3).

The evolution results for each state are discussed below. As was the case with the evolved multipliers, each evolution attempt was limited to 3000 generations.

6.2.1 State one

The final simplified phenotype is shown in Fig. 15. Within the first 39 generations, two of the four CPs were evolved, with the third and fourth CPs taking 1890 and 2964 generations respectively.

As expected, Fig. 15 shows why the first two CPs, $Pump_1$ and $Heater_1$, evolved quickly, the CPs only pass one LE. In contrast, $V_{Outlet2}$ and V_{Inlet2} pass five and six LEs respectively. In total, ten of the twenty available LEs were used

Notice that $Pump_1$ and $V_{Outlet1}$ are independent of each other, but dependent on V_{Inlet1}, i.e. $Pump_1$ and $V_{Outlet1}$ were independent CPs until V_{Inlet1} was evolved. The significance of this is discussed in Section 6.3.

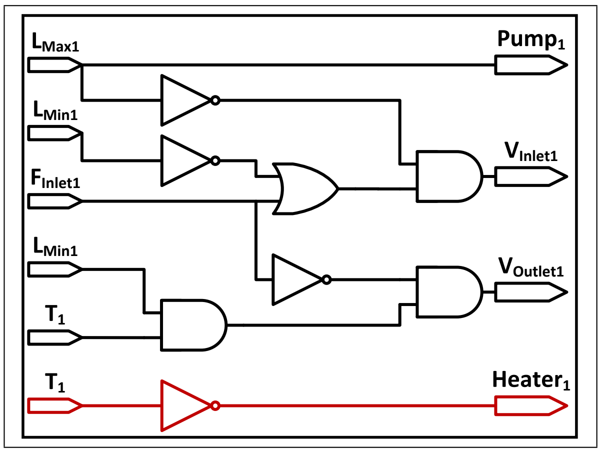


Figure 15: State one's simplified phenotype

6.2.2 State two

The evolved phenotype, shown in Fig. 16, makes use of six LEs. At fitness values 61.9%, 71%, 82.5% and 100%, the outputs $Heater_2$, $T_{L\_O\_2}$, $T_{S\_O\_2}$ and $V_{Outlet2}$ respectively were correctly evolved. The $Heater_2$, $T_{L\_O\_2}$ and $T_{S\_O\_2}$ CPs pass one LE while $V_{Outlet2}$ CP passes six. It took a total of 1085 generations to evolve the phenotype.

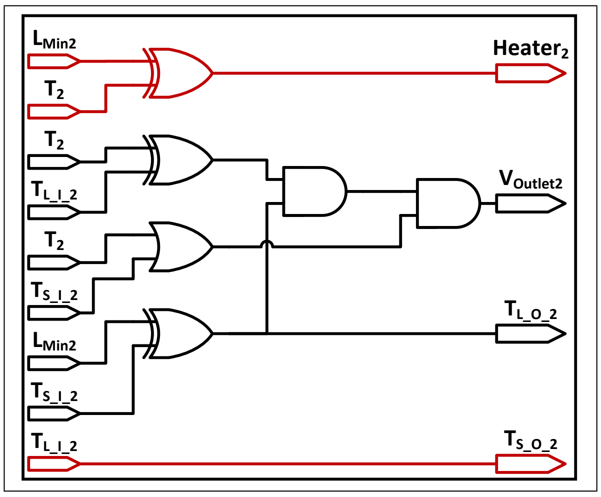


Figure 16: State two's simplified phenotype

6.3 Evolved FSM discussion

6.3.1 The evolved critical paths

As was done in Section 5.2.2, Table 3 shows the net number of LEs that were evolved for each CP. Unlike with the evolved multipliers, the state-one CPs were not all evolved in ascending order of the net number. In particular, $V_{Outlet1}$ was evolved before V_{Inlet1} even though their net numbers are five and four respectively. This is because the $Pump_1$ and $V_{Outlet1}$ CPs are independent of each other, but are both dependent on V_{Inlet1}.

During evolution, two or more initially independent CPs, or pre-independent CPs, can be linked together to form a third CP, thereby making all the CPs dependent. The third CP will take longer to evolve since the pre-independent CPs first need to be evolved; but will require fewer net LEs because the CP will be dependent. This explains why the V_{Inlet1} CP, although last to evolve, has a lower net-number.

All CPs with one net LE were evolved within the first 200 generations. In contrast, the $V_{Outlet1}$ and $V_{Outlet2}$ CPs both have a net number of five—the highest of all the CPs evolved in this study. Both took over 1000 generations to evolve, with the $V_{Outlet1}$ CP taking 1890 generations.

The above results are in concurrence with the results of the multiplier CPs in Section 5. The more net LEs a GA has to evolve, the longer the evolution takes (with the exception of pre-independent CPs).

From the simplified phenotypes, the complete combinational-logic circuit for the FSM can be realised by adjoining the sub-circuits.

7 LIMITATIONS AND FUTURE WORK

A number of important limitations of this study need to be considered. Firstly, while recognising that the hardware setup produced successful results, data communication between LabVIEW and the VRC was mostly slow, and at times inaccurate. This was evident in the spikes seen in Section 4 and the long evolution times. Future research will investigate completely eliminating LabVIEW and the DAQ interfacing hardware, and instead concentrate on implementing the GA using a soft-processor, thereby creating an on-chip solution.

The current study has only examined modular evolution using state decomposition, which relied on each state's sub-circuit being evolvable. However, this will not always be the case, as complex states will require further decomposition. Stomeo et al. have proposed a new method called 'Generalised Disjunction Decomposition', or GDD, where a circuit is decomposed according to the inputs [21]. This is based on the fact that the number of generations required to evolve a circuit is directly influenced by the number of external inputs.

Stomeo et al. showed that a 15-input circuit can take ten times longer to evolve than a 10-input circuit.

Future work should consider using GDD when evolving larger state sub-circuits, especially since the GDD research has shown usefulness in combinational-logic evolution. In addition, Stomeo et al. have concluded that GDD-evolved circuits have reached "higher values of fitness during optimisation", and are thus more efficient.

Finally, though recognising the results documented here and by others involved small and simple circuits, these circuits should not be dismissed, as they still play a major role in optimising system parameters. Small circuits are far more practical to analyse, and provide important insight into scalability and the unusual ways in which GAs synthesise circuits.

Until scalability is overcome, and evolution can provide solutions to real-world applications, further progress in the field will be required to make EHW a credible engineering tool.

Table 3: Summary of the net number of LEs evolved by each CP

	State one's phenotype				State two's phenotype			
	$Pump_1$	$Heater_1$	$V_{Outlet1}$	V_{Inlet1}	$Heater_2$	$T_{L\_O\_2}$	$T_{S\_O\_2}$	$V_{Outlet2}$
Number of evolved LEs used by the CP	1	1	5	6	1	1	1	6
Number of LEs used by the CP that were previously evolved by other CPs	0	0	0	2	0	0	0	1
Net number of LEs that were evolved (Net evolved LEs)	1	1	5	4	1	1	1	5

8 CONCLUSIONS

This paper investigated optimisation of EHW by means of digital circuit critical path analysis. The optimisation is done is by: omitting a canonical GAs crossover operator (i.e. by using a $1 + \lambda$ algorithm); applying evolution constraints; and optimising the fitness function. A 2×2 digital multiplier and a state-decomposed control circuit were evolved using a target-independent VRC architecture.

The results showed that the evolved multiplier circuits, when compared to a conventional multiplier, are either equal or more efficient. All the evolved circuits improve two of the four critical paths, and all are unique.

By comparing the $1 + \lambda$ and canonical GAs, the results verify that optimised GAs can find solutions quicker, and with fewer attempts. Part of the optimisation includes a comprehensive critical-path analysis, where the findings show that the identification of dependent critical paths is vital in enhancing a GA's efficiency.

A critical-path analysis of each circuit needs to be completed in order to understand the manner in which the circuits are evolved. With this knowledge, better fitness functions and operators was developed to further enhance the EHW system's efficiency.

9 ACKNOWLEDGMENTS

The financial assistance of the National Research Foundation (NRF) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at are those of the authors and are not necessarily to be attributed to the NRF.

The authors would like to thank the anonymous reviewers for their invaluable feedback.

REFERENCES

[1] G. T. Gordon and J. P. Bentley. "On evolvable hardware". In S. Ovaska and L. Sztandera (editors), *Soft computing in industrial electronics*, pp. 279–323. Springer, 2002. DOI `http://dx.doi.org/10.1007/978-3-7908-1783-6_8`.

[2] F. Cancare, S. Bhandari, D. B. Bartolini, M. Carminati and M. D. Santambrogio. "A bird's eye view of FPGA-based evolvable hardware". In *2011 NASA/ESA conference on adaptive hardware and systems (AHS)*, pp. 169–175. IEEE, 2011. DOI `http://dx.doi.org/10.1109/AHS.2011.5963932`.

[3] E. Stomeo, T. Kalganova and C. Lambert. "Mutation rate for evolvable hardware." In *International conference on computational intelligence - ICCI 2005*, pp. 117–124. 2005.

[4] P. Husbands, R. C. Moioli, Y. Shim, A. Philippides, P. A. Vargas and M. O'Shea. "Evolutionary robotics and neuroscience". In *The horizons of evolutionary robotics*, chap. 2, pp. 17–63. MIT Press, March 2014.

[5] A. Stoica, R. Zebulum, D. Keymeulen, M. Ferguson and X. Guo. "Scalability issues in evolutionary synthesis of electronic circuits: Lessons learned and challenges ahead". In *American association for artificial intelligence*. 2003.

[6] T. Higuchi, M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, N. Salami, N. Kajihara et al. "Real-world applications of analog and digital evolvable hardware". *IEEE transactions on evolutionary computation*, vol. 3, no. 3, pp. 220–235, 1999. DOI `http://dx.doi.org/10.1109/4235.788492`.

[7] M. Matarić and D. Cliff. "Challenges in evolving controllers for physical robots". *Robotics and autonomous systems*, vol. 19, no. 1, pp. 67–83, 1996. DOI `http://dx.doi.org/10.1016/S0921-8890(96)00034-6`.

[8] H. de Garis, N. Nawa, F. Gers, M. Korkin and A. Agah. "'CAM-brain' ATR's billion neuron artificial brain project: A three-year progress report". *Artificial life and robotics*, vol. 2, no. 2, pp. 56–61, 1998. DOI `http://dx.doi.org/10.1007/BF02471155`.

[9] A. Bedi. *A generic platform for the evolution of hardware.* Ph.D. thesis, Auckland University of Technology, 2009.

[10] L. Sekanina. "FPGA-based evolvable hardware systems". In *Proceedings of the 26th international conference on architecture of computing systems, February 2013, Prague, Czech Republic.* 2013.

[11] L. Sekanina and Š. Friedl. "An evolvable combinational unit for FPGAs". *Computing and informatics*, vol. 23, no. 5-6, pp. 461–486, 2012.

[12] J. Torresen. "An evolvable hardware tutorial". In *Field-programmable logic and application*, pp. 821–830. Springer, 2004. DOI `http://dx.doi.org/10.1007/978-3-540-30117-2_83`.

[13] F. Smith. "A virtual VLSI architecture for computer hardware evolution". In *Proceedings of the 2010 annual research conference of the South African institute of computer scientists and information technolo-*

gists (SAICSIT), pp. 294–303. ACM, 2010. DOI `http://dx.doi.org/10.1145/1899503.1899536`.

[14] T. G. Gordon and P. J. Bentley. "Development brings scalability to hardware evolution". In *Proceedings of the 2005 NASA/DoD conference on evolvable hardware.*, pp. 272–279. IEEE, 2005. DOI `http://dx.doi.org/10.1109/eh.2005.18`.

[15] A. Swarnalatha and A. Shanthi. "Complete hardware evolution based SoPC for evolvable hardware". *Applied soft computing*, vol. 18, pp. 314–322, 2014. DOI `http://dx.doi.org/10.1016/j.asoc.2013.12.014`.

[16] R. Dobai and L. Sekanina. "Towards evolvable systems based on the Xilinx Zynq platform". In *2013 IEEE international conference on evolvable systems (ICES)*, pp. 89–95. IEEE, 2013. DOI `http://dx.doi.org/10.1109/ICES.2013.6613287`.

[17] J. Wang, C. H. Piao and C. H. Lee. "Implementing multi-VRC cores to evolve combinational logic circuits in parallel". In *Evolvable systems: From biology to hardware*, pp. 23–34. Springer, 2007. DOI `http://dx.doi.org/10.1007/978-3-540-74626-3_3`.

[18] P. Martin and R. Poli. "Crossover operators for a hardware implementation of GP using FPGAs and Handel-C". In *GECCO*, pp. 845–852. 2002.

[19] V. K. Vassilev, J. F. Miller and T. C. Fogarty. "On the nature of two-bit multiplier landscapes". In *Proceedings of the first NASA/DoD workshop on evolvable hardware, 1999.*, pp. 36–45. IEEE, 1999. DOI `http://dx.doi.org/10.1109/eh.1999.785433`.

[20] T. Kalganova. "Bidirectional incremental evolution in extrinsic evolvable hardware". In *Proceedings of the second NASA/DoD workshop on evolvable hardware, 2000.*, pp. 65–74. IEEE, 2000.

[21] E. Stomeo, T. Kalganova and C. Lambert. "Generalized disjunction decomposition for evolvable hardware". *IEEE transactions on systems, man, and cybernetics, part B: cybernetics*, vol. 36, no. 5, pp. 1024–1043, 2006.

[22] A. Thompson. "An evolved circuit, intrinsic in silicon, entwined with physics". In *Evolvable systems: From biology to hardware*, pp. 390–405. Springer, 1997.

[23] F. Cancare, M. D. Santambrogio and D. Sciuto. "A direct bitstream manipulation approach for Virtex4-based evolvable systems". In *Proceedings of 2010 IEEE international symposium on circuits and systems (ISCAS)*, pp. 853–856. IEEE, 2010.

[24] M. Majzoobi, F. Koushanfar and M. Potkonjak. "Trusted design in FPGAs". In *Introduction to hardware security and trust*, pp. 195–229. Springer, 2012. DOI `http://dx.doi.org/10.1007/978-1-4419-8080-9_9`.

[25] C. Slorach and K. Sharman. "The design and implementation of custom architectures for evolvable hardware using off-the-shelf programmable devices". In *Evolvable systems: From biology to hardware*, pp. 197–207. Springer, 2000. DOI `http://dx.doi.org/10.1007/3-540-46406-9_20`.

[26] L. Sekanina and A. M. Sllame. "Toward uniform approach to design of evolvable hardware based systems". In *Field-programmable logic and applications: The roadmap to reconfigurable computing*, pp. 814–817. Springer, 2000. DOI `http://dx.doi.org/10.1007/3-540-44614-1_92`.

[27] L. Altenberg. "The evolution of evolvability in genetic programming". In K. Kinnear (editor), *Advances in genetic programming*, vol. 3, pp. 47–74. Cambridge, MA, 1994.